Tell us about your PDF experience.

Install C and C++ support in Visual Studio

Article • 12/09/2021

If you haven't downloaded and installed Visual Studio and the Microsoft C/C++ tools yet, here's how to get started.

Visual Studio 2022 Installation

Welcome to Visual Studio 2022! In this version, it's easy to choose and install just the features you need. And because of its reduced minimum footprint, it installs quickly and with less system impact.

> **Note**
>
> This topic applies to installation of Visual Studio on Windows. **Visual Studio Code** is a lightweight, cross-platform development environment that runs on Windows, Mac, and Linux systems. The Microsoft **C/C++ for Visual Studio Code** extension supports IntelliSense, debugging, code formatting, auto-completion. Visual Studio for Mac doesn't support Microsoft C++, but does support .NET languages and cross-platform development. For installation instructions, see **Install Visual Studio for Mac**.

Want to know more about what else is new in this version? See the Visual Studio release notes.

Ready to install? We'll walk you through it, step-by-step.

Step 1 - Make sure your computer is ready for Visual Studio

Before you begin installing Visual Studio:

1. Check the system requirements. These requirements help you know whether your computer supports Visual Studio 2022.
2. Apply the latest Windows updates. These updates ensure that your computer has both the latest security updates and the required system components for Visual Studio.

3. Reboot. The reboot ensures that any pending installs or updates don't hinder the Visual Studio install.

4. Free up space. Remove unneeded files and applications from your %SystemDrive% by, for example, running the Disk Cleanup app.

For questions about running previous versions of Visual Studio side by side with Visual Studio 2022, see the Visual Studio 2022 Platform Targeting and Compatibility page.

Step 2 - Download Visual Studio

Next, download the Visual Studio bootstrapper file. To do so, choose the following button to go to the Visual Studio download page. Select the edition of Visual Studio that you want and choose the **Free trial** or **Free download** button.

Download Visual Studio

Step 3 - Install the Visual Studio installer

Run the bootstrapper file you downloaded to install the Visual Studio Installer. This new lightweight installer includes everything you need to both install and customize Visual Studio.

1. From your **Downloads** folder, double-click the bootstrapper that matches or is similar to one of the following files:

 - **vs_community.exe** for Visual Studio Community
 - **vs_professional.exe** for Visual Studio Professional
 - **vs_enterprise.exe** for Visual Studio Enterprise

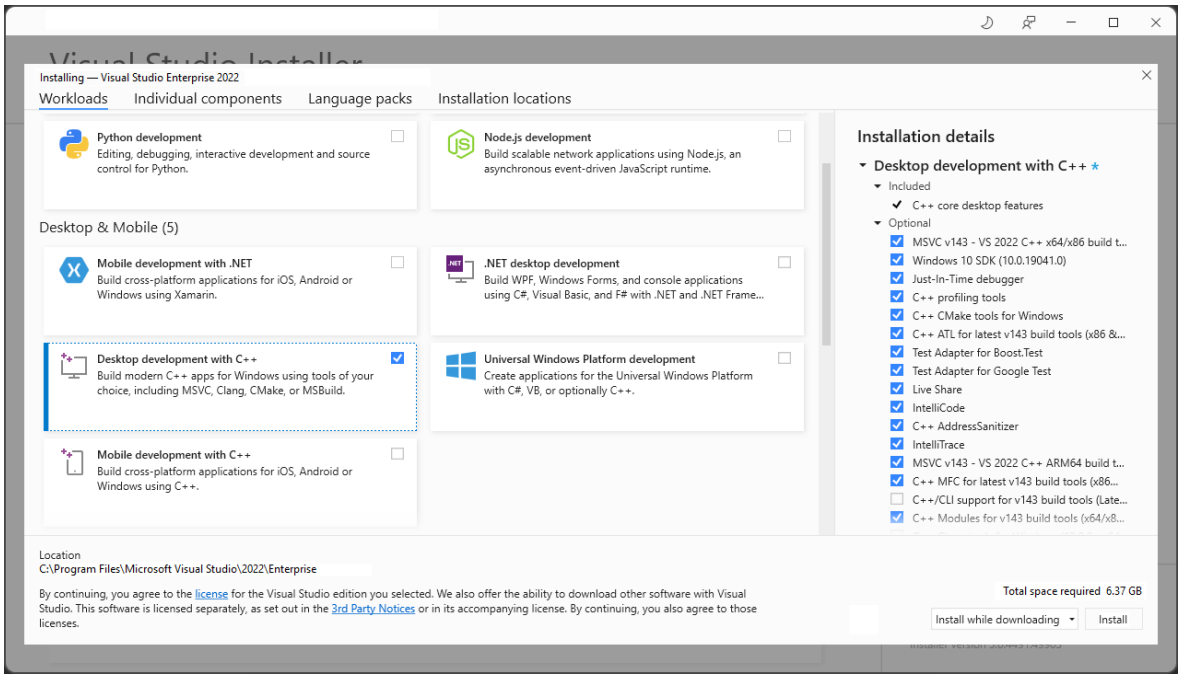
 If you receive a User Account Control notice, choose **Yes** to allow the bootstrapper to run.

2. We'll ask you to acknowledge the Microsoft License Terms and the Microsoft Privacy Statement. Choose **Continue**.

Step 4 - Choose workloads

After the installer is installed, you can use it to customize your installation by selecting the *workloads*, or feature sets, that you want. Here's how.

1. Find the workload you want in the **Installing Visual Studio** screen.



For core C and C++ support, choose the "Desktop development with C++" workload. It comes with the default core editor, which includes basic code editing support for over 20 languages, the ability to open and edit code from any folder without requiring a project, and integrated source code control.

Additional workloads support other kinds of development. For example, choose the "Universal Windows Platform development" workload to create apps that use the Windows Runtime for the Microsoft Store. Choose "Game development with C++" to create games that use DirectX, Unreal, and Cocos2d. Choose "Linux development with C++" to target Linux platforms, including IoT development.

The **Installation details** pane lists the included and optional components installed by each workload. You can select or deselect optional components in this list. For example, to support development by using the Visual Studio 2017 or 2015 compiler toolsets, choose the MSVC v141 or MSVC v140 optional components. You can add support for MFC, the experimental Modules language extension, IncrediBuild, and more.

2. After you choose the workload(s) and optional components you want, choose **Install**.

 Next, status screens appear that show the progress of your Visual Studio installation.

Tip

At any time after installation, you can install workloads or components that you didn't install initially. If you have Visual Studio open, go to **Tools** > **Get Tools and**

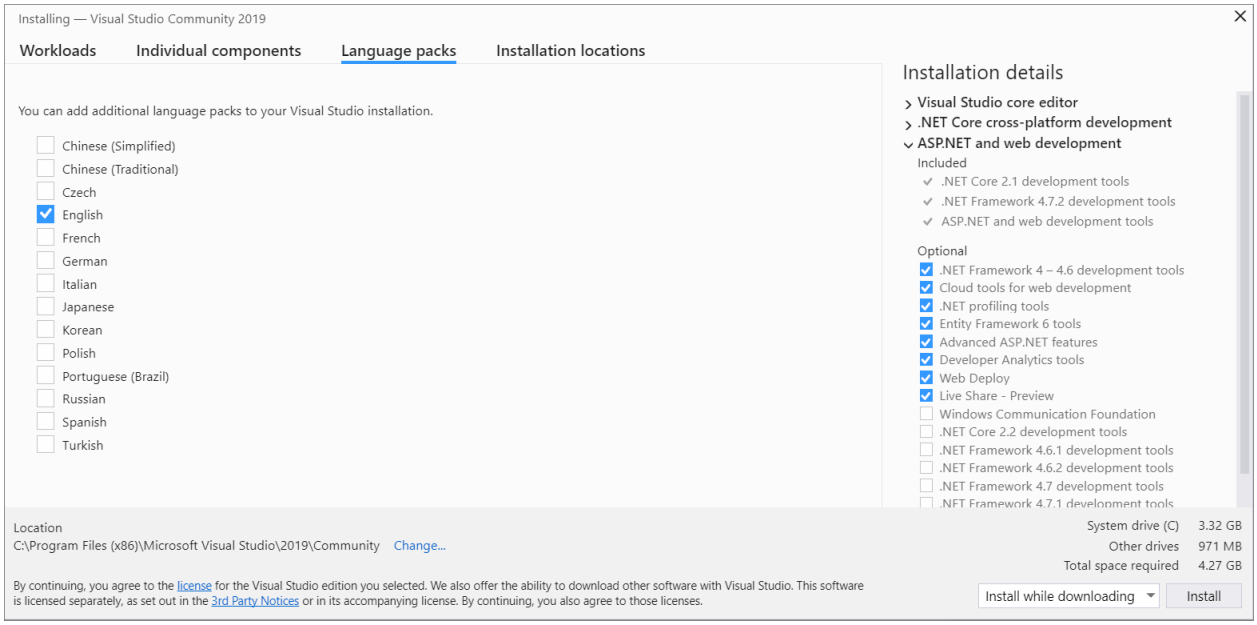
Features... which opens the Visual Studio Installer. Or, open **Visual Studio Installer** from the Start menu. From there, you can choose the workloads or components that you wish to install. Then, choose **Modify**.

Step 5 - Choose individual components (Optional)

If you don't want to use the Workloads feature to customize your Visual Studio installation, or you want to add more components than a workload installs, you can do so by installing or adding individual components from the **Individual components** tab. Choose what you want, and then follow the prompts.

Step 6 - Install language packs (Optional)

By default, the installer program tries to match the language of the operating system when it runs for the first time. To install Visual Studio in a language of your choosing, choose the **Language packs** tab from the Visual Studio Installer, and then follow the prompts.



Change the installer language from the command line

Another way that you can change the default language is by running the installer from the command line. For example, you can force the installer to run in English by using the following command: `vs_installer.exe --locale en-US`. The installer will remember this setting when it's run the next time. The installer supports the following language tokens: zh-cn, zh-tw, cs-cz, en-us, es-es, fr-fr, de-de, it-it, ja-jp, ko-kr, pl-pl, pt-br, ru-ru, and tr-tr.

Step 7 - Change the installation location (Optional)

You can reduce the installation footprint of Visual Studio on your system drive. You can choose to move the download cache, shared components, SDKs, and tools to different drives, and keep Visual Studio on the drive that runs it the fastest.

> **ⓘ Important**
>
> You can select a different drive only when you first install Visual Studio. If you've already installed it and want to change drives, you must uninstall Visual Studio and then reinstall it.

Step 8 - Start developing

1. After Visual Studio installation is complete, choose the **Launch** button to get started developing with Visual Studio.

2. On the start window, choose **Create a new project**.

3. In the search box, enter the type of app you want to create to see a list of available templates. The list of templates depends on the workload(s) that you chose during installation. To see different templates, choose different workloads.

 You can also filter your search for a specific programming language by using the **Language** drop-down list. You can filter by using the **Platform** list and the **Project type** list, too.

4. Visual Studio opens your new project, and you're ready to code!

When Visual Studio is running, you're ready to continue to the next step.

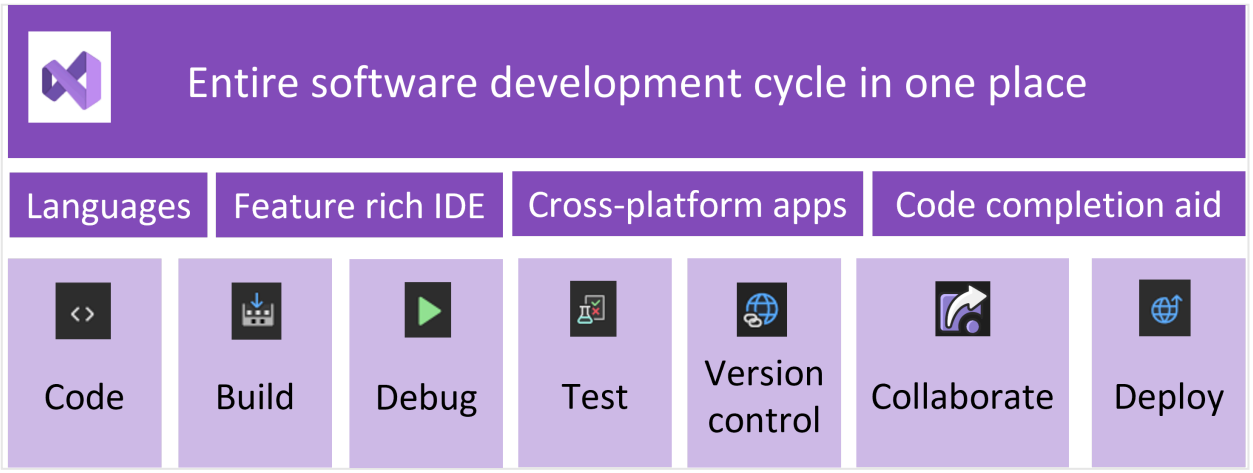
Next Steps

Create a C++ project

What is Visual Studio?

Article • 06/19/2024

Visual Studio is a powerful developer tool that you can use to complete the entire development cycle in one place. It's a comprehensive integrated development environment (IDE) that you can use to write, edit, debug, and build code. Then deploy your app. Visual Studio includes compilers, code completion tools, source control, extensions, and many other features to enhance every stage of the software development process.



With the variety of features and languages support in Visual Studio, you can grow from writing your first "Hello World" program to developing and deploying apps. For example, build, debug, and test .NET and C++ apps, edit ASP.NET pages in the web designer view, develop cross-platform mobile and desktop apps with .NET, or build responsive Web UIs in C#.

To install Visual Studio, select the following button, and choose the edition of Visual Studio to download.

Download Visual Studio

Why use Visual Studio?

Visual Studio provides developers a feature rich development environment to develop high-quality code efficiently and collaboratively.

- Workload-based installer - install only what you need
- Powerful coding tools and features - everything you need to build your apps in one place

- Multiple language support - code in C++, C#, JavaScript, TypeScript, Python, and more
- Cross-platform development - build apps for any platform
- Version control integration - collaborate on code with team mates
- AI-assisted development - write code more efficiently with AI assistance

Discover Visual Studio

Visual Studio supports different parts of the software development cycle.

Develop your code

Visual Studio IDE provides many features that make it easier for you to write and manage your code with confidence. For example, code quickly and accurately with AI-assisted development tools. These tools include GitHub Copilot and IntelliCode. Make quick improvements to your code using light bulbs that suggest actions, or expand/collapse blocks of code using outlining. Organize and explore your code with the **Solution Explorer** that shows your code organized by files or the **Class View** that shows your code organized by classes.

Learn more about all the features in the IDE that help you organize and edit content:

- Code editor
- Personalize the IDE and the editor
- Organize code
- Tips and tricks

AI-assisted development

GitHub Copilot, GitHub Copilot Chat, and IntelliCode assist developers in writing code faster and with greater accuracy, help develop a deeper understanding of the codebase, and help with other development tasks such as writing unit tests, debugging, and profiling.

Learn more about AI-assisted development in Visual Studio:

Get started with GitHub Copilot in Visual Studio:

- Install and manage GitHub Copilot
- Use GitHub Copilot Completions in Visual Studio
- Use GitHub Copilot Chat in Visual Studio
- Debug with Copilot

Build your app

You can compile and build your applications to create builds immediately and test them in a debugger. You can run multi-processor builds for C++ and C# projects. Visual Studio also provides several options that you can configure when you build applications. You can create a custom build configuration in addition to the built-in configurations, hide certain warning messages, or increase build output information.

Learn more about how to compile and build in Visual Studio:

- Create build configurations for your project
- Build an application

Debug your code

Integrated debugging in Visual Studio enables you to debug, profile, and diagnose with ease. You step through your code and look at the values stored in variables, set watches on variables to see when values changes, examine the execution path of your code. Visual Studio offers other ways to debug your code while it runs.

Learn more about debugging effectively in Visual Studio:

- Debug your app
- Debugging techniques and tools
- Measure app performance
- Debug with Copilot
- Tips and tricks

Test your code

You can write high-quality code with comprehensive testing tools in Visual Studio. Unit tests give developers and testers a quick way to find logic errors in code. You can analyze how much code you're testing and see instant results in a test suite. Know the impact of every change you make with advanced features that test code while you type.

Learn more about the testing tools available in Visual Studio:

- Use testing tools in Visual Studio
- Create and run unit tests
- Analyze code coverage

Version control

With the integrated Git features in Visual Studio, you can clone, create, or open your own repositories. The Git tool window has everything you need to commit and push changes, manage branches, and resolve merge conflicts. If you have a GitHub account, you can manage those repos directly within Visual Studio.

Learn more about version control in Visual Studio:

- Version control with Git
- Visual Studio and GitHub

Collaborate with others

Visual Studio Live Share enables real-time collaborative development. With Live Share you can share your project with your peers, no matter the language or platform. Get to the bottom of an issue fast by allowing your team to connect, navigate, set break points, and type in your editor session.

Learn more about how to collaborate with Live Share:

- Collaborate with Live Share
- Common use cases

Deploy your app

By deploying an application, service, or component, you distribute it for installation on other computers, devices, or servers, or in the cloud. You can choose the appropriate method in Visual Studio for the type of deployment that you need. Share your apps and code by publishing to the web or Azure, or by deploying to a network share or a local folder.

Learn more about how to deploy your app using Visual Studio:

- Deploy your app from Visual Studio
- Deploy your app to a folder, a web server, Azure, or another destination

Choose your Visual Studio edition

There are three editions of Visual Studio:

- Community - free, fully featured IDE for students, open-source developers, and individual developers.
- Professional - a subscription based option for individual developers or small teams.

- Enterprise - a subscription based option for small to large business and enterprise organizations.

Compare features across Visual Studio editions and acquire the Visual Studio edition that best fits your needs.

Install Visual Studio

Select the following button to install Visual Studio, and choose the edition of Visual Studio.

Download Visual Studio

Get started

Dive into coding with one of the following language-specific tutorials:

- Create a simple C# console app
- Get started with Python
- Create a simple VB console app
- Create a C++ console app
- Create a Node.js and Express app

Related content

To develop any type of app, or learn a language, you work in the feature rich Visual Studio Integrated Development Environment (IDE). Explore Visual Studio further with one of these introductory articles:

- Tour the IDE to get familiar with the IDE features and to learn how to use it for basic tasks.
- Cover the basics in this Learn module: Introduction to Visual Studio

Feedback

Was this page helpful? Yes No

Provide product feedback | Ask the community

Create a C++ console app project

Article • 07/06/2023

The usual starting point for a C++ programmer is a "Hello, world!" application that runs on the command line. That's what you create in Visual Studio in this step.

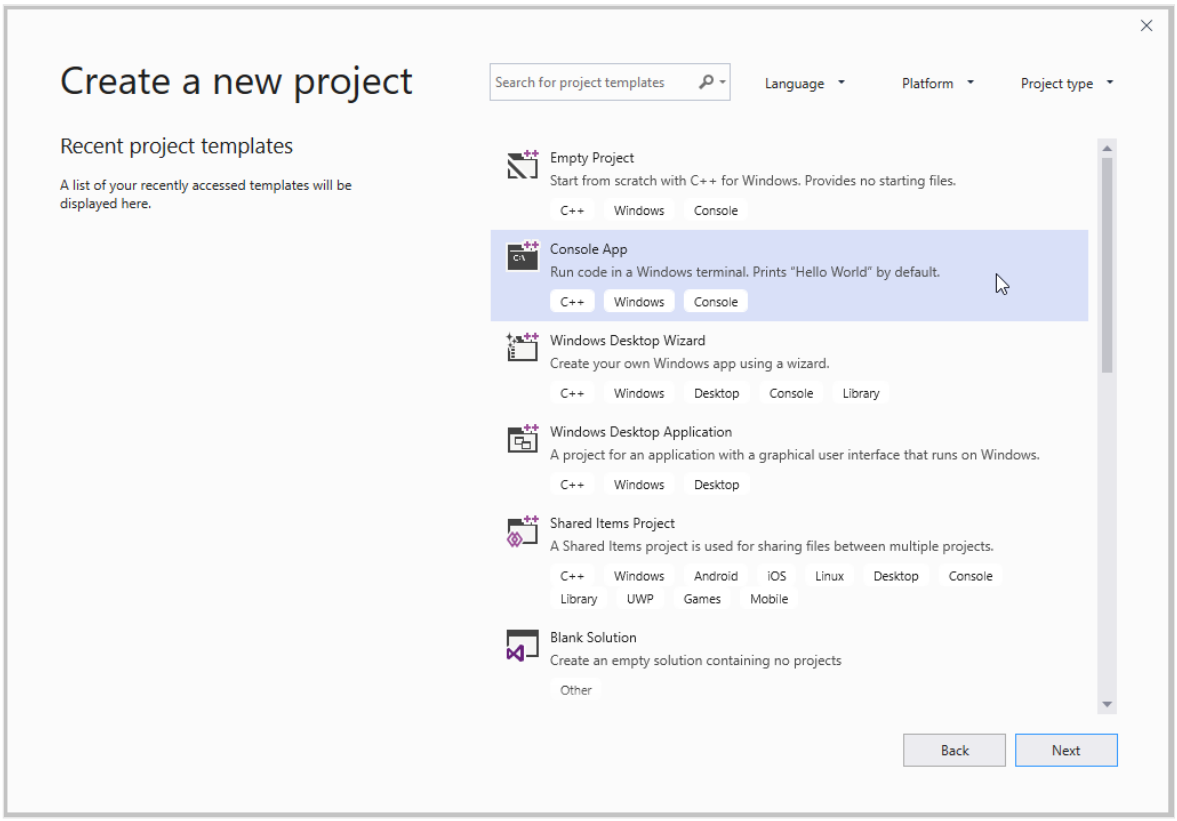
Prerequisites

- Have Visual Studio with the Desktop development with C++ workload installed and running on your computer. If it's not installed yet, see Install C++ support in Visual Studio.

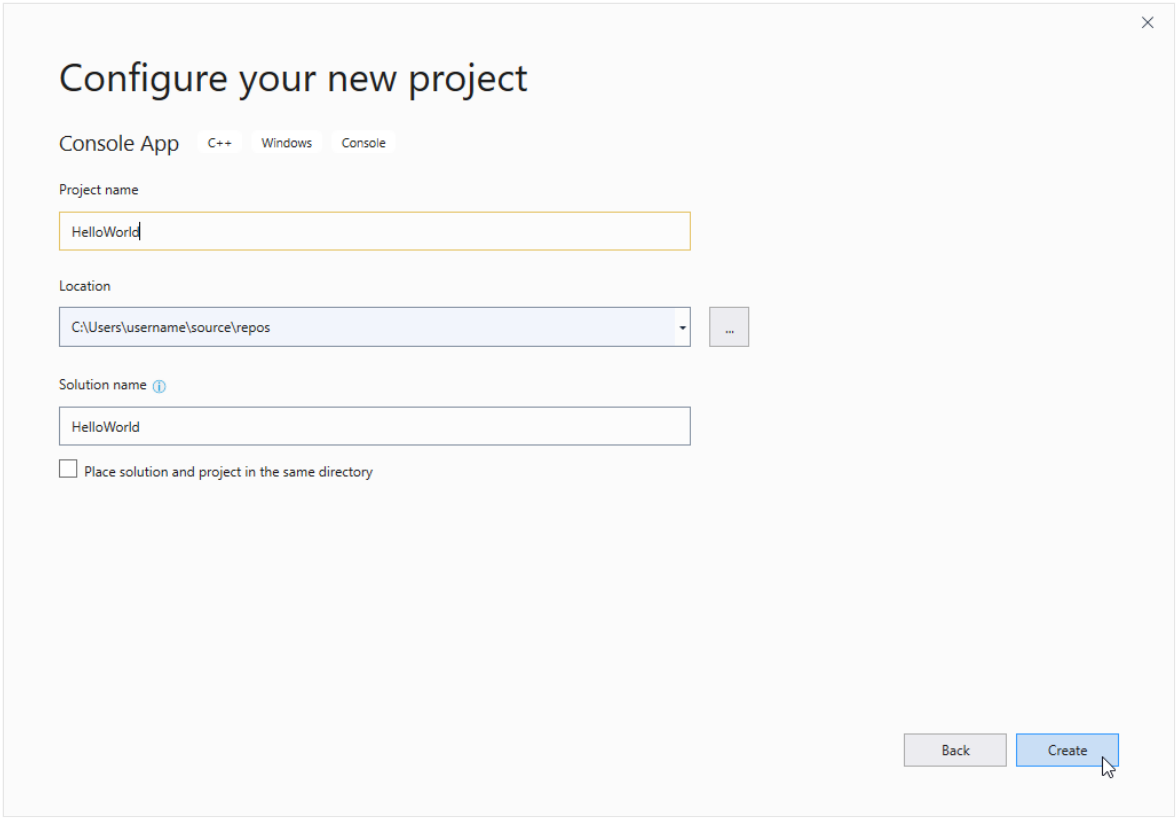
Create your app project

Visual Studio uses *projects* to organize the code for an app, and *solutions* to organize your projects. A project contains all the options, configurations, and rules used to build your apps. It manages the relationship between all the project's files and any external files. To create your app, first, create a new project and solution.

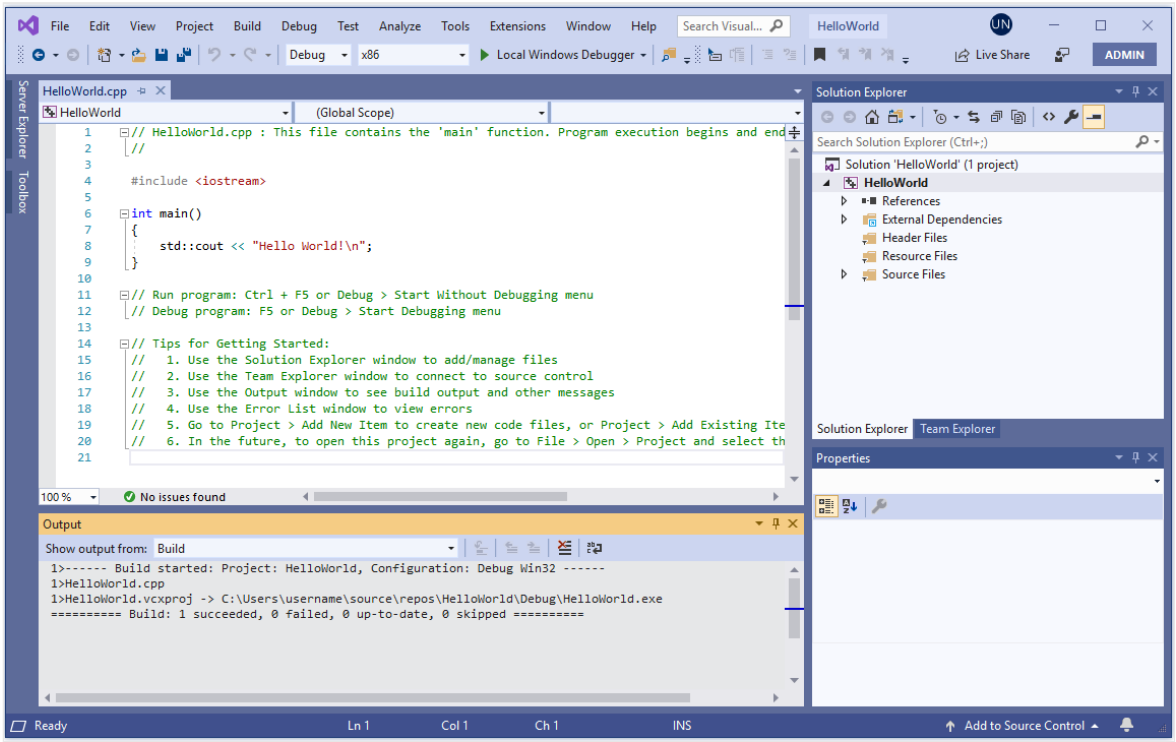
1. In Visual Studio, open the **File** menu and choose **New** > **Project** to open the **Create a new Project** dialog. Select the **Console App** template that has **C++**, **Windows**, and **Console** tags, and then choose **Next**.



2. In the **Configure your new project** dialog, enter *HelloWorld* in the **Project name** edit box. Choose **Create** to create the project.



Visual Studio creates a new project. It's ready for you to add and edit your source code. By default, the Console App template provides source code for a "Hello World" app, like this:



When the code looks like this in the editor, you're ready to go on to the next step and build your app.

I ran into a problem.

Next steps

Build and run a C++ project

Troubleshooting guide

Come here for solutions to common issues when you create your first C++ project.

Create your app project: issues

The **New Project** dialog should show a **Console App** template that has **C++**, **Windows**, and **Console** tags. If you don't see it, there are two possible causes. It might be filtered out of the list, or it might not be installed. First, check the filter dropdowns at the top of the list of templates. Set them to **C++**, **Windows**, and **Console**. The C++ **Console App** template should appear; otherwise, the **Desktop development with C++** workload isn't installed.

To install **Desktop development with C++**, you can run the installer right from the **New Project** dialog. Choose the **Install more tools and features** link at the bottom of the template list to start the installer. If the **User Account Control** dialog requests permissions, choose **Yes**. In the installer, make sure the **Desktop development with C++** workload is checked. Then choose **Modify** to update your Visual Studio installation.

If another project with the same name already exists, choose another name for your project. Or, delete the existing project and try again. To delete an existing project, delete the solution folder (the folder that contains the `helloworld.sln` file) in File Explorer.

Go back.

Build and run a C++ console app project

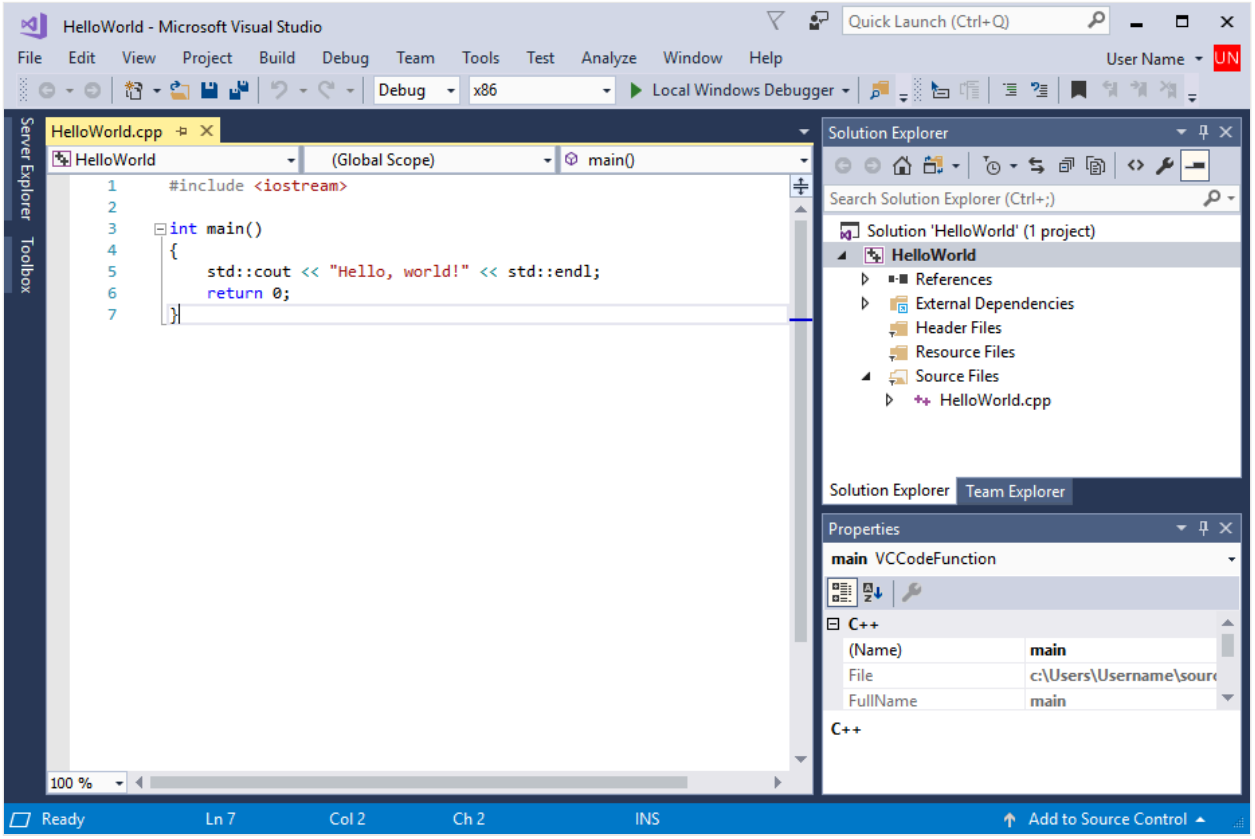
Article • 07/01/2024

In Create a C++ console app project you created a C++ console app project and entered your code. Now you can build and run it within Visual Studio. Then, run it as a stand-alone app from the command line.

Prerequisites

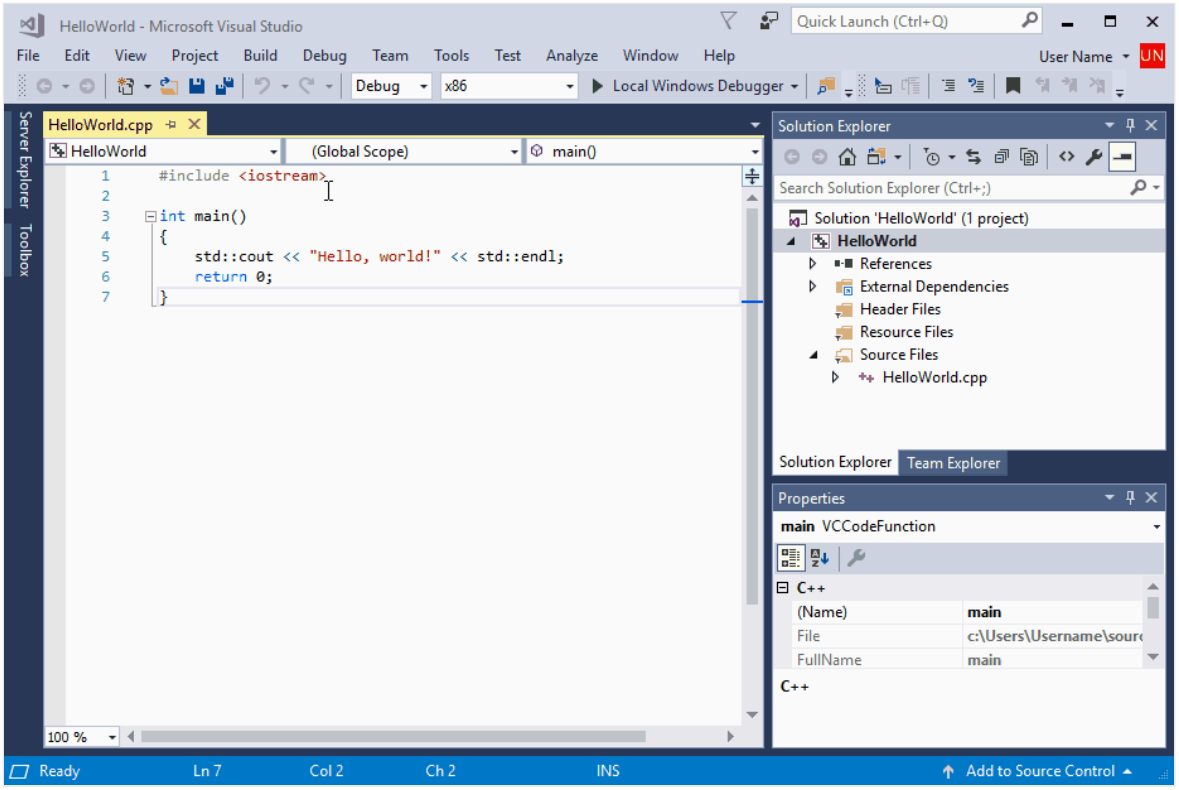
- Have Visual Studio with the Desktop development with C++ workload installed and running on your computer. If it's not installed, follow the steps in Install C++ support in Visual Studio.
- Create a "Hello, World!" project. By default, it contains code to print `Hello World!`. If you haven't done this step yet, follow the steps in Create a C++ console app project.

If Visual Studio looks like this, you're ready to build and run your app:

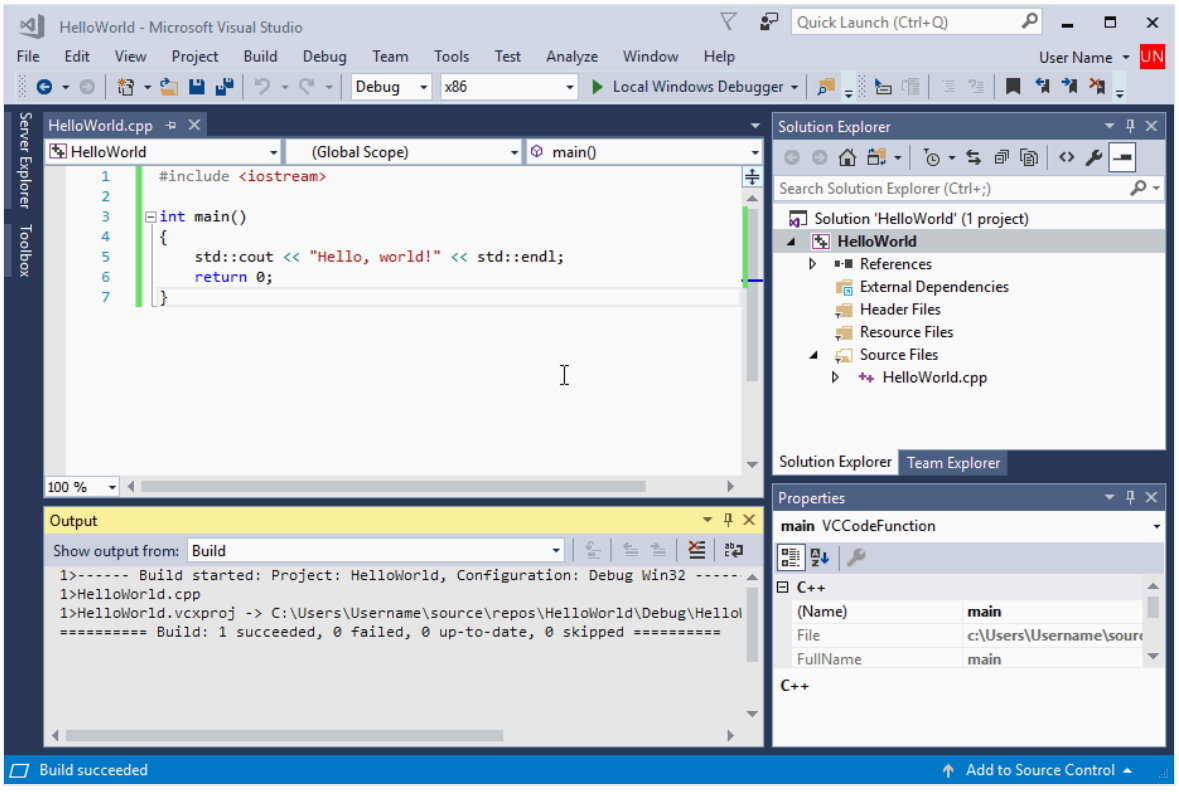


Build and run your code in Visual Studio

1. To build your project, from the main menu choose **Build** > **Build Solution**. The **Output** window shows the results of the build process.



2. To run the code, on the menu bar, choose **Debug**, **Start without debugging**.



A console window opens and then runs your app. When you start a console app in Visual Studio, it runs your code, then prints "Press any key to continue . . ." to give you a chance to see the output.

Congratulations! You created your first "Hello, world!" console app in Visual Studio! Press a key to dismiss the console window and return to Visual Studio.

I ran into a problem.

Run your code in a command window

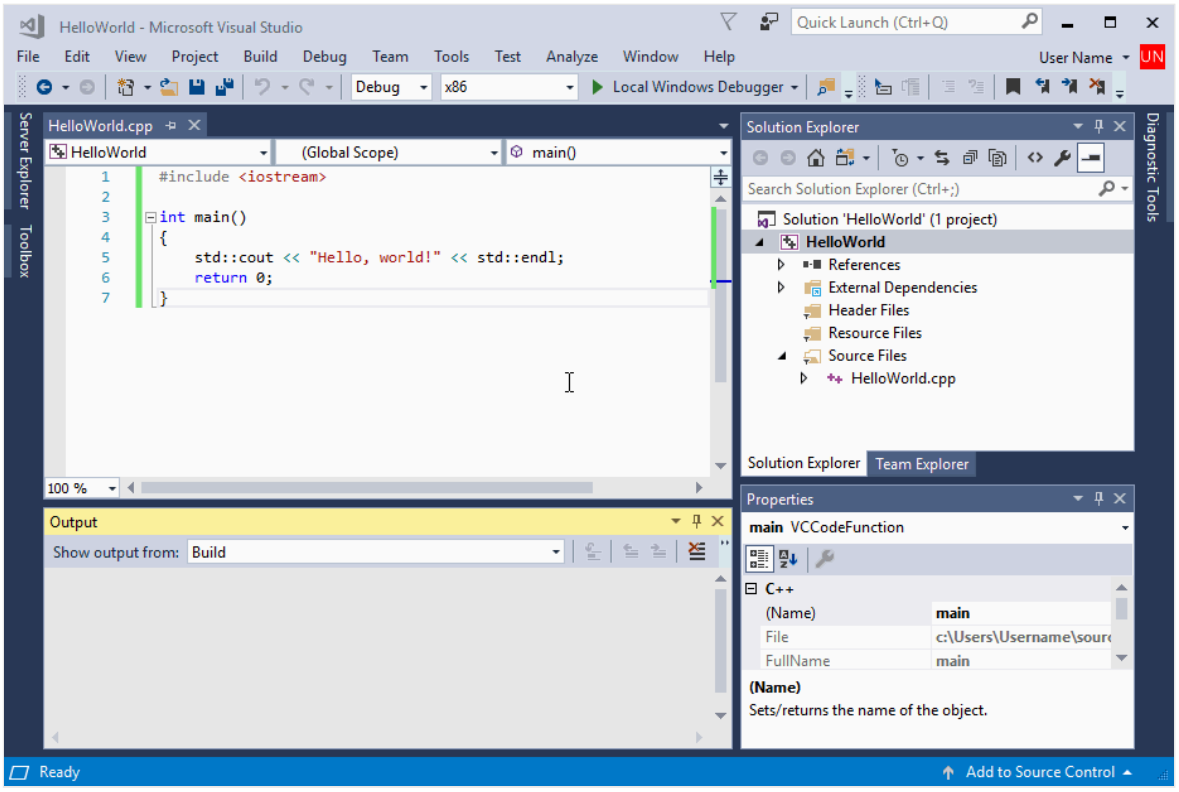
Normally, you run console apps at the command prompt, not in Visual Studio. Once Visual Studio builds your app, you can run it from a command window. Here's how to find and run your new app in a command prompt window.

1. In **Solution Explorer**, select the HelloWorld solution (not the HelloWorld project) and right-click to open the context menu. Choose **Open Folder in File Explorer** to open a **File Explorer** window in the HelloWorld solution folder.

2. In the **File Explorer** window, open the `x64` folder and then the `Debug` folder. This folder contains your app, `HelloWorld.exe`, and debugging files. Hold down the **Shift** key and right-click on `HelloWorld.exe` to open the context menu. Choose **Copy as path** to copy the path to your app to the clipboard. If you see `HelloWorld.exe.recipe`, it's because you did the **Open Folder in File Explorer** step on the HelloWorld *project* instead of the HelloWorld *solution*. Navigate up a level in File Explorer to get to the solution folder. This folder also contains a `x64\Debug\` folder, where `HelloWorld.exe` is.

3. To open a command prompt window, press **Windows+R** to open the **Run** dialog. Enter *cmd.exe* in the **Open** textbox, then choose **OK** to run a command prompt window.

4. In the command prompt window, right-click to paste the path to your app into the command prompt. Press Enter to run your app.



Congratulations, you built and ran a console app in Visual Studio!

I ran into a problem.

Next Steps

Once you build and run this simple app, you're ready for more complex projects. For more information, see Using the Visual Studio IDE for C++ Desktop Development. It has more detailed walkthroughs that explore the capabilities of Microsoft C++ in Visual Studio.

Troubleshooting guide

Come here for solutions to common issues when you create your first C++ project.

Build and run your code in Visual Studio: issues

If red squiggles appear under anything in the source code editor, the build may have errors or warnings. Check that your code matches the example in spelling, punctuation, and case.

Go back.

Run your code in a command window: issues

If the path shown in File Explorer ends in `\HelloWorld\HelloWorld`, you opened the HelloWorld *project* instead of the HelloWorld *solution*. You won't see your app in the `x64\Debug` folder. Navigate up a level in File Explorer to get to the solution folder, the first `HelloWorld` in the path. This folder also contains a `x64\Debug` folder, where your app is.

You can also navigate to the solution `x64\Debug` folder at the command line to run your app. Your app won't run from other directories without specifying the path to the app. However, you can copy your app to another directory and run it from there. It's also possible to copy it to a directory specified by your `PATH` environment variable, then run it from anywhere.

If you don't see **Copy as path** in the shortcut menu, dismiss the menu, and then hold down the **Shift** key while you open it again. This command is just for convenience. You can also copy the path to the folder from the File Explorer search bar, and paste it into the **Run** dialog, and then enter the name of your executable at the end. It's just a little more typing, but it has the same result.

Go back.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Welcome back to C++ - Modern C++

Article • 11/07/2022

Since its creation, C++ has become one of the most widely used programming languages in the world. Well-written C++ programs are fast and efficient. The language is more flexible than other languages: It can work at the highest levels of abstraction, and down at the level of the silicon. C++ supplies highly optimized standard libraries. It enables access to low-level hardware features, to maximize speed and minimize memory requirements. C++ can create almost any kind of program: Games, device drivers, HPC, cloud, desktop, embedded, and mobile apps, and much more. Even libraries and compilers for other programming languages get written in C++.

One of the original requirements for C++ was backward compatibility with the C language. As a result, C++ has always permitted C-style programming, with raw pointers, arrays, null-terminated character strings, and other features. They may enable great performance, but can also spawn bugs and complexity. The evolution of C++ has emphasized features that greatly reduce the need to use C-style idioms. The old C-programming facilities are still there when you need them. However, in modern C++ code you should need them less and less. Modern C++ code is simpler, safer, more elegant, and still as fast as ever.

The following sections provide an overview of the main features of modern C++. Unless noted otherwise, the features listed here are available in C++11 and later. In the Microsoft C++ compiler, you can set the /std compiler option to specify which version of the standard to use for your project.

Resources and smart pointers

One of the major classes of bugs in C-style programming is the *memory leak*. Leaks are often caused by a failure to call `delete` for memory that was allocated with `new`. Modern C++ emphasizes the principle of *resource acquisition is initialization* (RAII). The idea is simple. Resources (heap memory, file handles, sockets, and so on) should be *owned* by an object. That object creates, or receives, the newly allocated resource in its constructor, and deletes it in its destructor. The principle of RAII guarantees that all resources get properly returned to the operating system when the owning object goes out of scope.

To support easy adoption of RAII principles, the C++ Standard Library provides three smart pointer types: std::unique_ptr, std::shared_ptr, and std::weak_ptr. A smart pointer handles the allocation and deletion of the memory it owns. The following example

shows a class with an array member that is allocated on the heap in the call to `make_unique()`. The calls to `new` and `delete` are encapsulated by the `unique_ptr` class. When a `widget` object goes out of scope, the unique_ptr destructor will be invoked and it will release the memory that was allocated for the array.

C++

```
#include <memory>
class widget
{
private:
    std::unique_ptr<int[]> data;
public:
    widget(const int size) { data = std::make_unique<int[]>(size); }
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000);  // lifetime automatically tied to enclosing scope
                        // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data
```

Whenever possible, use a smart pointer to manage heap memory. If you must use the `new` and `delete` operators explicitly, follow the principle of RAII. For more information, see Object lifetime and resource management (RAII).

`std::string` and `std::string_view`

C-style strings are another major source of bugs. By using std::string and std::wstring, you can eliminate virtually all the errors associated with C-style strings. You also gain the benefit of member functions for searching, appending, prepending, and so on. Both are highly optimized for speed. When passing a string to a function that requires only read-only access, in C++17 you can use std::string_view for even greater performance benefit.

`std::vector` and other Standard Library containers

The standard library containers all follow the principle of RAII. They provide iterators for safe traversal of elements. And, they're highly optimized for performance and have been thoroughly tested for correctness. By using these containers, you eliminate the potential

for bugs or inefficiencies that might be introduced in custom data structures. Instead of raw arrays, use vector as a sequential container in C++.

```C++
vector<string> apples;
apples.push_back("Granny Smith");
```

Use map (not `unordered_map`) as the default associative container. Use set, multimap, and multiset for degenerate and multi cases.

```C++
map<string, string> apple_color;
// ...
apple_color["Granny Smith"] = "Green";
```

When performance optimization is needed, consider using:

- Unordered associative containers such as unordered_map. These have lower per-element overhead and constant-time lookup, but they can be harder to use correctly and efficiently.
- Sorted `vector`. For more information, see Algorithms.

Don't use C-style arrays. For older APIs that need direct access to the data, use accessor methods such as `f(vec.data(), vec.size());` instead. For more information about containers, see C++ Standard Library Containers.

Standard Library algorithms

Before you assume that you need to write a custom algorithm for your program, first review the C++ Standard Library algorithms. The Standard Library contains an ever-growing assortment of algorithms for many common operations such as searching, sorting, filtering, and randomizing. The math library is extensive. In C++17 and later, parallel versions of many algorithms are provided.

Here are some important examples:

- `for_each`, the default traversal algorithm (along with range-based `for` loops).
- `transform`, for not-in-place modification of container elements
- `find_if`, the default search algorithm.
- `sort`, `lower_bound`, and the other default sorting and searching algorithms.

To write a comparator, use strict `<` and use *named lambdas* when you can.

C++

```
auto comp = [](const widget& w1, const widget& w2)
     { return w1.weight() < w2.weight(); }

sort( v.begin(), v.end(), comp );

auto i = lower_bound( v.begin(), v.end(), widget{0}, comp );
```

`auto` instead of explicit type names

C++11 introduced the auto keyword for use in variable, function, and template declarations. **`auto`** tells the compiler to deduce the type of the object so that you don't have to type it explicitly. **`auto`** is especially useful when the deduced type is a nested template:

C++

```
map<int,list<string>>::iterator i = m.begin(); // C-style
auto i = m.begin(); // modern C++
```

Range-based `for` loops

C-style iteration over arrays and containers is prone to indexing errors and is also tedious to type. To eliminate these errors, and make your code more readable, use range-based **`for`** loops with both Standard Library containers and raw arrays. For more information, see Range-based for statement.

C++

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {1,2,3};

    // C-style
    for(int i = 0; i < v.size(); ++i)
    {
        std::cout << v[i];
    }
```

```
    // Modern C++:
    for(auto& num : v)
    {
        std::cout << num;
    }
}
```

`constexpr` expressions instead of macros

Macros in C and C++ are tokens that are processed by the preprocessor before compilation. Each instance of a macro token is replaced with its defined value or expression before the file is compiled. Macros are commonly used in C-style programming to define compile-time constant values. However, macros are error-prone and difficult to debug. In modern C++, you should prefer constexpr variables for compile-time constants:

C++

```
#define SIZE 10 // C-style
constexpr int size = 10; // modern C++
```

Uniform initialization

In modern C++, you can use brace initialization for any type. This form of initialization is especially convenient when initializing arrays, vectors, or other containers. In the following example, `v2` is initialized with three instances of `S`. `v3` is initialized with three instances of `S` that are themselves initialized using braces. The compiler infers the type of each element based on the declared type of `v3`.

C++

```
#include <vector>

struct S
{
    std::string name;
    float num;
    S(std::string s, float f) : name(s), num(f) {}
};

int main()
{
    // C-style initialization
    std::vector<S> v;
    S s1("Norah", 2.7);
```

```
    S s2("Frank", 3.5);
    S s3("Jeri", 85.9);

    v.push_back(s1);
    v.push_back(s2);
    v.push_back(s3);

    // Modern C++:
    std::vector<S> v2 {s1, s2, s3};

    // or...
    std::vector<S> v3{ {"Norah", 2.7}, {"Frank", 3.5}, {"Jeri", 85.9} };

}
```

For more information, see Brace initialization.

Move semantics

Modern C++ provides *move semantics*, which make it possible to eliminate unnecessary memory copies. In earlier versions of the language, copies were unavoidable in certain situations. A *move* operation transfers ownership of a resource from one object to the next without making a copy. Some classes own resources such as heap memory, file handles, and so on. When you implement a resource-owning class, you can define a *move constructor* and *move assignment operator* for it. The compiler chooses these special members during overload resolution in situations where a copy isn't needed. The Standard Library container types invoke the move constructor on objects if one is defined. For more information, see Move Constructors and Move Assignment Operators (C++).

Lambda expressions

In C-style programming, a function can be passed to another function by using a *function pointer*. Function pointers are inconvenient to maintain and understand. The function they refer to may be defined elsewhere in the source code, far away from the point at which it's invoked. Also, they're not type-safe. Modern C++ provides *function objects*, classes that override the operator() operator, which enables them to be called like a function. The most convenient way to create function objects is with inline lambda expressions. The following example shows how to use a lambda expression to pass a function object, that the `find_if` function will invoke on each element in the vector:

```
C++
```

```
    std::vector<int> v {1,2,3,4,5};
    int x = 2;
    int y = 4;
    auto result = find_if(begin(v), end(v), [=](int i) { return i > x && i <
y; });
```

The lambda expression `[=](int i) { return i > x && i < y; }` can be read as "function that takes a single argument of type `int` and returns a boolean that indicates whether the argument is greater than `x` and less than `y`." Notice that the variables `x` and `y` from the surrounding context can be used in the lambda. The `[=]` specifies that those variables are *captured* by value; in other words, the lambda expression has its own copies of those values.

Exceptions

Modern C++ emphasizes exceptions, not error codes, as the best way to report and handle error conditions. For more information, see Modern C++ best practices for exceptions and error handling.

`std::atomic`

Use the C++ Standard Library std::atomic struct and related types for inter-thread communication mechanisms.

`std::variant` (C++17)

Unions are commonly used in C-style programming to conserve memory by enabling members of different types to occupy the same memory location. However, unions aren't type-safe and are prone to programming errors. C++17 introduces the std::variant class as a more robust and safe alternative to unions. The std::visit function can be used to access the members of a `variant` type in a type-safe manner.

See also

C++ Language Reference
Lambda Expressions
C++ Standard Library
Microsoft C/C++ language conformance

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Create a console calculator in C++

Article • 10/08/2024

The usual starting point for a C++ programmer is a "Hello, world!" application that runs on the command line. You start with that in this article, and then move on to something more challenging: a calculator app.

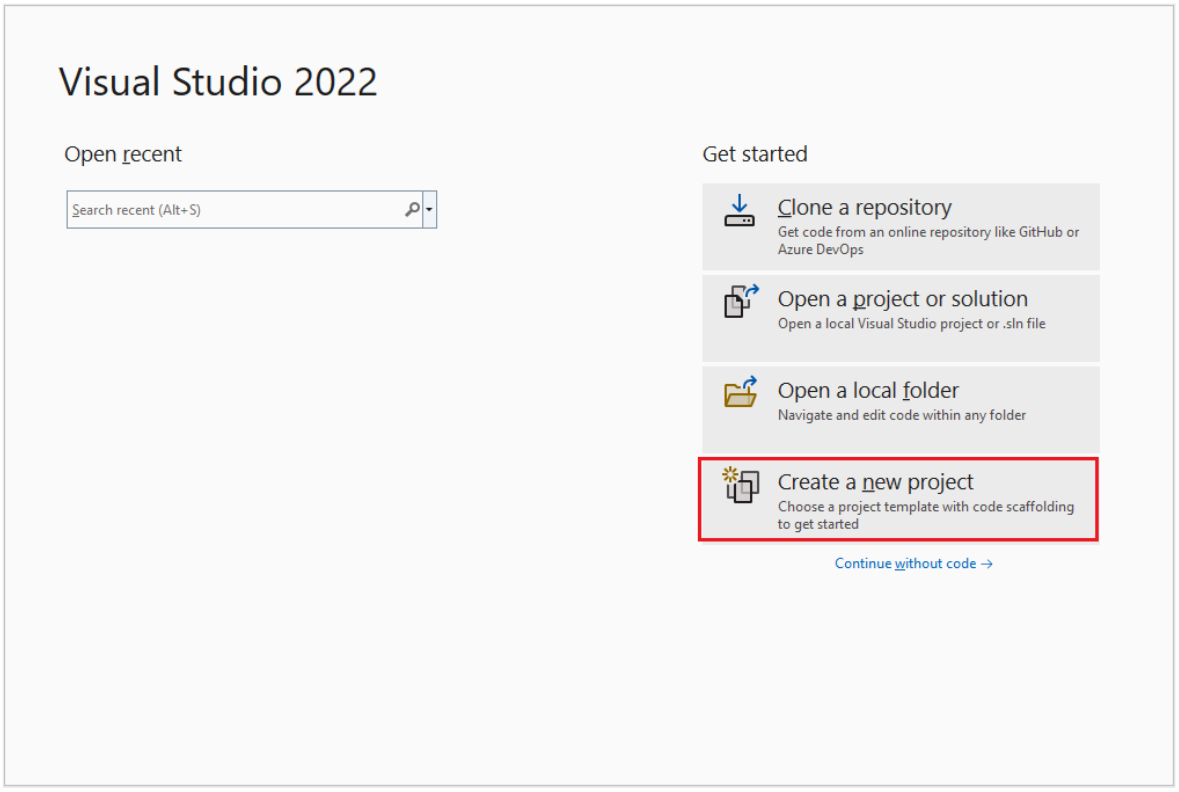
Prerequisites

- Visual Studio with the **Desktop development with C++** workload installed and running on your computer. To install it, see Install C++ support in Visual Studio.
- This tutorial demonstrates a feature called edit and continue which allows you to make changes to your code while the app is running. To enable edit and continue, from the main menu select **Tools** > **Options** > **Debugging** > **General** and ensure that **Require source files to exactly match the original version** is checked.

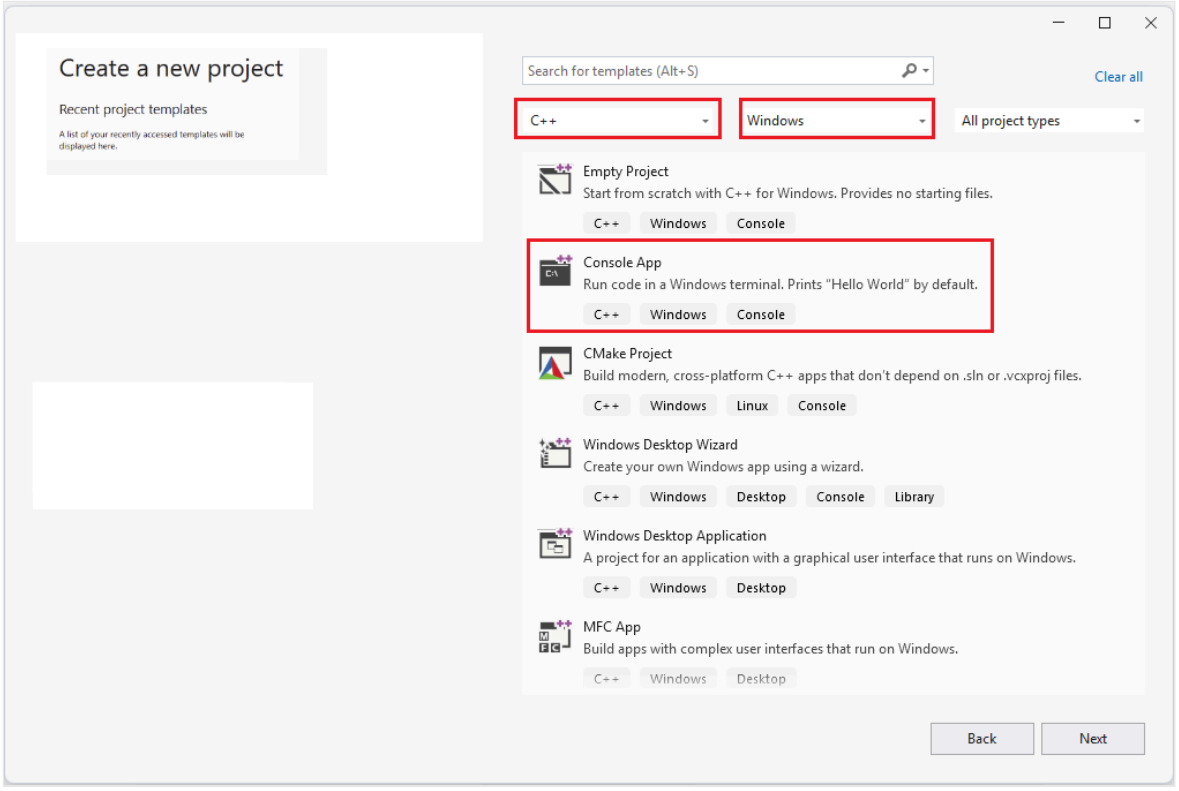
Create your app project

Visual Studio uses *projects* to organize the code for an app, and *solutions* to organize one or more projects. A project contains all the options, configurations, and rules used to build an app. It also manages the relationship between all the project's files and any external files. To create your app, first, create a new project and solution.

1. Start Visual Studio--the Visual Studio Start dialog box appears. Select **Create a new project** to get started.

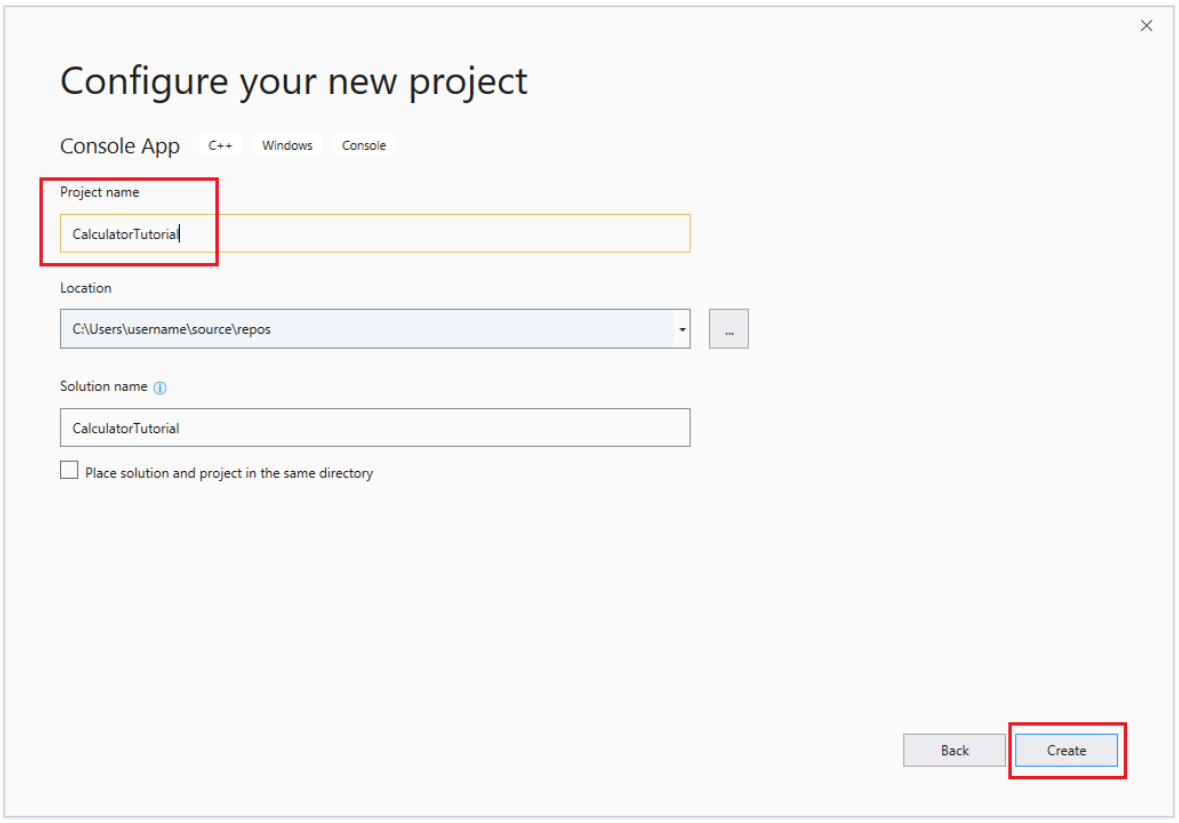


2. In the **Create a new project** dialog, set the language dropdown to **C++**, set the platform dropdown to **Windows**, select **Console App** from the list of project types, then select **Next**.



> **ⓘ Important**
>
> Make sure you select the C++ version of the **Console App** template. It has the **C++**, **Windows**, and **Console** tags, and the icon has "++" in the corner.

3. In the **Configure your new project** dialog box, select the **Project name** text box, name your new project *CalculatorTutorial*, then select **Create**.



An empty C++ Windows console application 'Hello World' app is created. Console applications use a Windows console window to display output and accept user input. In Visual Studio, an editor window opens and shows the generated code:

C++

```
// CalculatorTutorial.cpp : This file contains the 'main' function.
Program execution begins and ends there.
//

#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

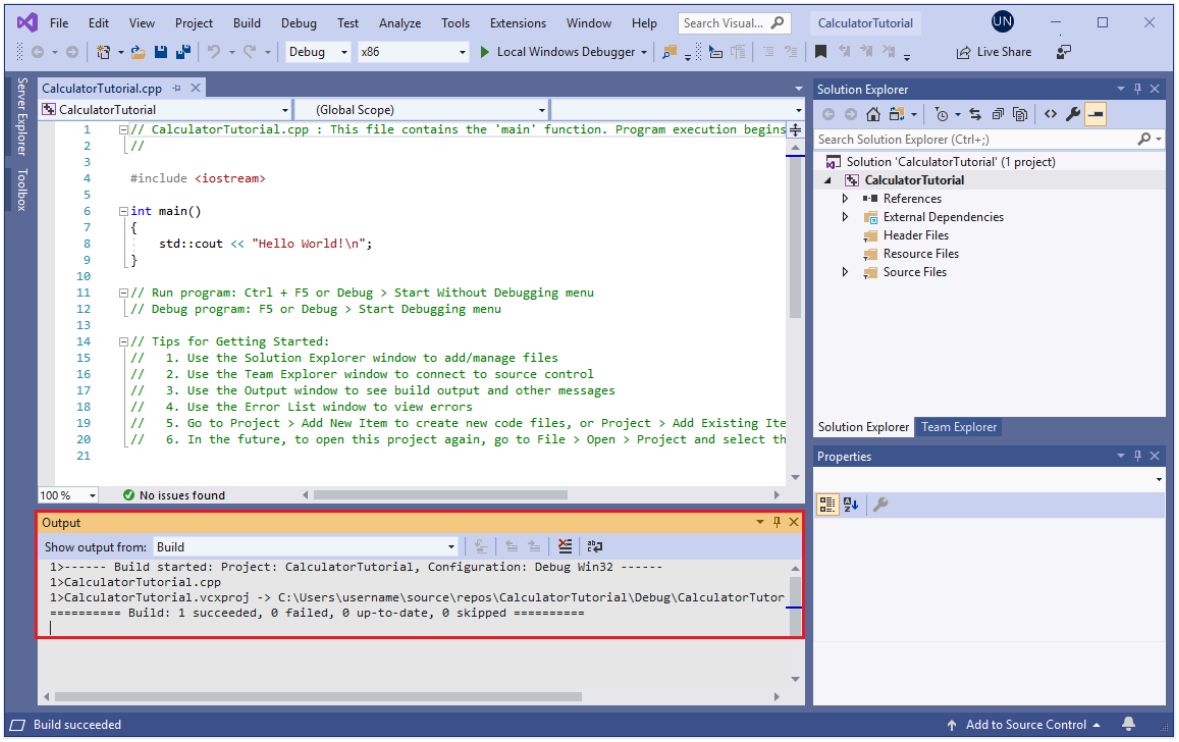
// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or
Project > Add Existing Item to add existing code files to the project
```

```
//   6. In the future, to open this project again, go to File > Open >
Project and select the .sln file
```

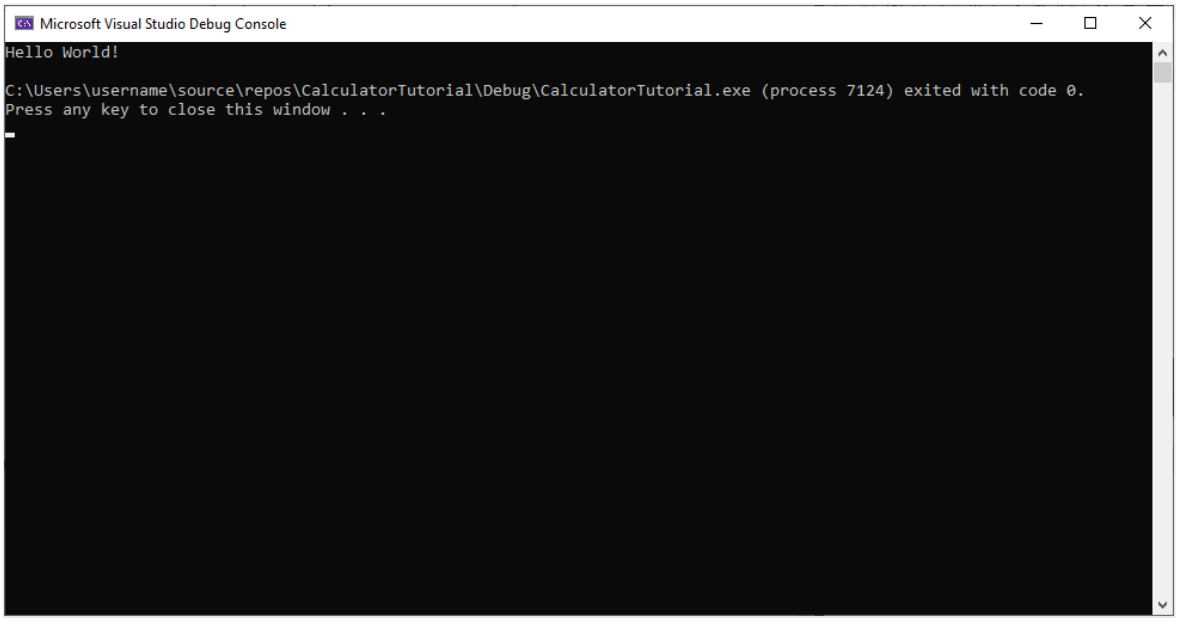
Verify that your new app builds and runs

The template for a new Windows console application creates a simple C++ "Hello World" app. At this point, you can see how Visual Studio builds and runs the apps you create right from the IDE.

1. To build your project, select **Build Solution** from the **Build** menu. The **Output** window shows the results of the build process.



2. To run the code, on the menu bar, select **Debug** > **Start without debugging** (Ctrl+F5).



A console window opens and your app runs within it.

When you start a console app in Visual Studio, it runs your code, then prints "Press any key to close this window . . ." to give you a chance to see the output.

Congratulations! You created your first "Hello, world!" console app in Visual Studio!

3. Press a key to dismiss the console window and return to Visual Studio.

You now have the tools to build and run your app after every change, to verify that the code still works as you expect. Later, we show you how to debug it if it doesn't.

Edit the code

Now let's modify the code in this template to be a calculator app.

1. Replace the contents of the *`CalculatorTutorial.cpp`* file with the following code so that it matches this example:

```
C++
```

```cpp
// CalculatorTutorial.cpp : This file contains the 'main' function.
Program execution begins and ends there.
//

#include <iostream>

using namespace std;

int main()
{
    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b |
```

```
a*b | a/b"
        << endl;
    return 0;
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu
// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or
Project > Add Existing Item to add existing code files to the project
//   6. In the future, to open this project again, go to File > Open >
Project and select the .sln file
```

Understanding the code:

- The `#include` statement brings in code in other files. Sometimes, you may see a filename surrounded by angle brackets like `<iostream>`. The angle brackets instruct the compiler to look for the `iostream` header file first in the standard system directories, and if not found, to look in directories specific to the project. Other times, you may see a filename surrounded by quotes like `"someHeader.h"`. The quotes instruct the compiler to skip looking in the standard system directories and instead only look in directories specific to the project.
- The `using namespace std;` tells the compiler to expect code from the C++ Standard Library to be used in this file. Without this line, each keyword from the library would have to be preceded with `std::` to denote its scope. For instance, without that line, each reference to `cout` would be written as `std::cout`. The **`using`** statement is added to make it more convenient to access code in another namespace.
- The `cout` keyword is used to print to standard output in C++. The `<<` operator tells the compiler to send whatever is to the right of it to the standard output.
- The `endl` keyword is like the Enter key; it ends the line and moves the cursor to the next line. It's a better practice to put a `\n` inside the string (contained by `""`) to do the same thing because `endl` always flushes the buffer which can hurt the performance of the program. But since this is a very small app, `endl` is used instead.
- All C++ statements must end with semicolons and all C++ applications must contain a `main()` function. This function is what the program runs at

> the start. All code must be accessible from `main()` in order to be used.

2. To save the file, press **Ctrl+S**, or select the floppy disk icon in the toolbar under the menu bar.

3. To run the application, press **Ctrl+F5** or go to the **Debug** menu and select **Start Without Debugging**. You should see a console window appear that looks like this.

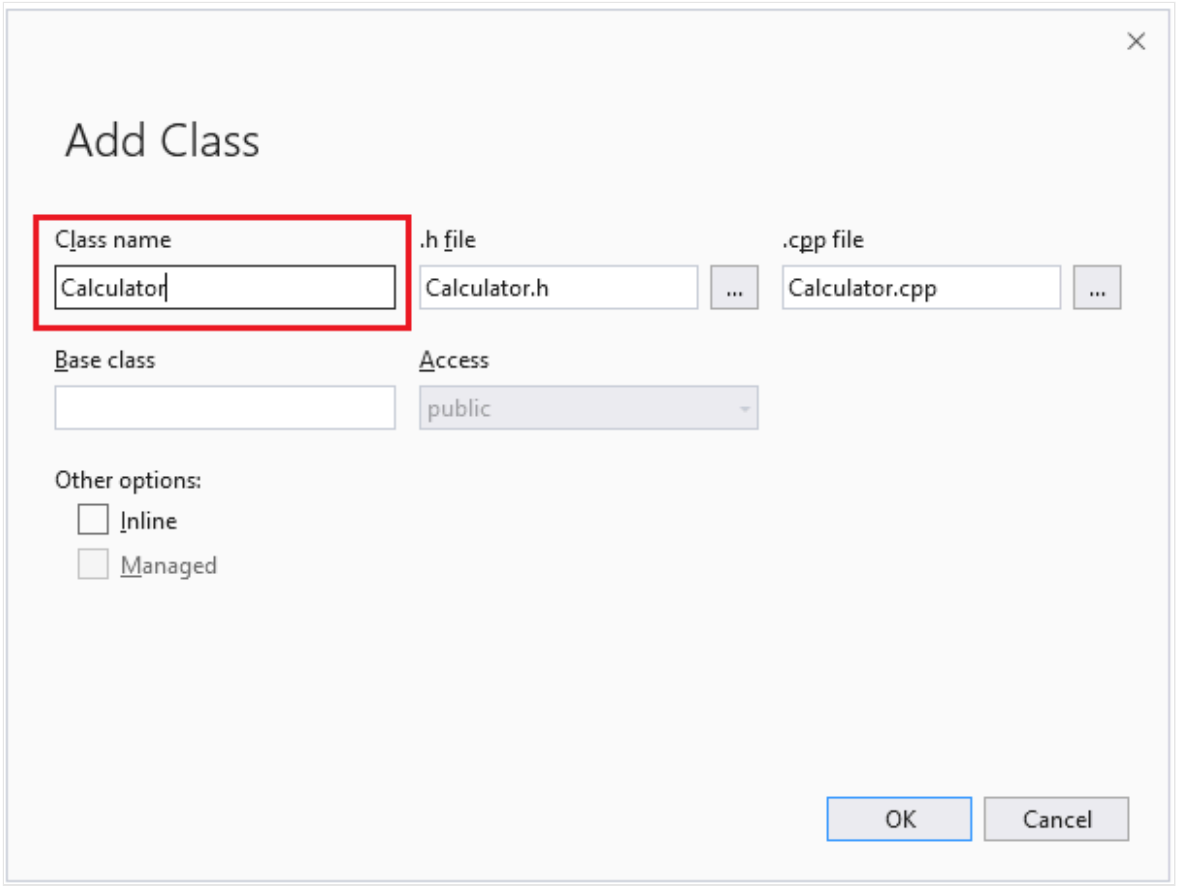
4. Close the console window when you're done.

Add code to do some math

A class is like a blueprint for an object that does something. In this case, we define a calculator class to contain the math logic.

Add a Calculator class

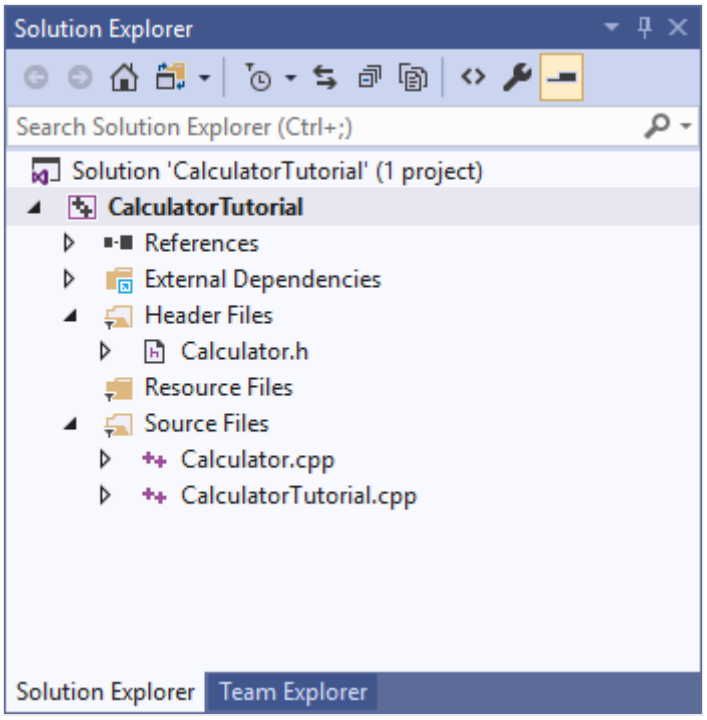
1. Go to the **Project** menu and select **Add Class**. In the **Class Name** edit box, enter *Calculator*. Select **OK**.



Two new files get added to your project. To save all your changed files at once, press **Ctrl+Shift+S**. It's a keyboard shortcut for **File** > **Save All**. There's also a toolbar button for **Save All**, an icon of two floppy disks, found beside the **Save**

button. In general, it's good practice to do **Save All** frequently, so you don't miss saving any changes.

The **Add Class** wizard creates `.h` and `.cpp` files that have the same name as the class. You can see a full list of your project files in the **Solution Explorer** window, visible on the side of the IDE. If the window isn't visible, open it from the menu bar via **View** > **Solution Explorer**.



You can open a file by double-clicking it in the **Solution Explorer** window. Double-click `Calculator.h` to open it.

2. Replace the contents of **`Calculator.h`** with the following code so that the file now looks like this:

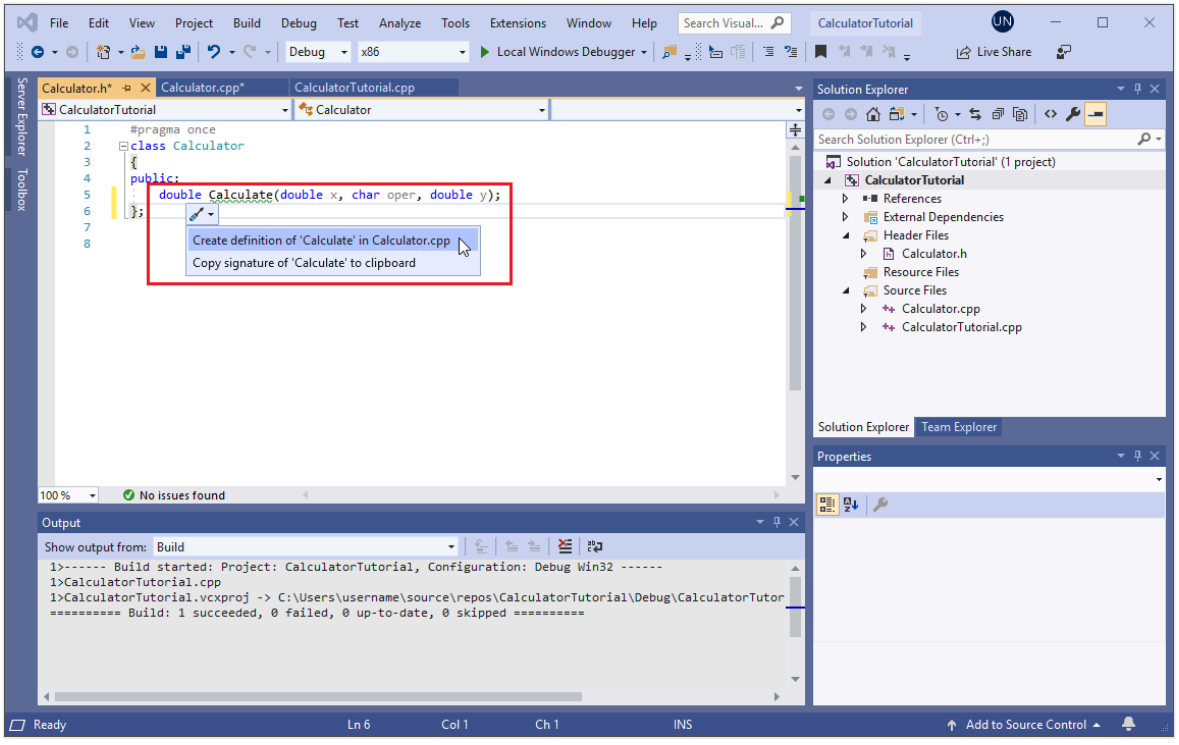
```
C++
```

```
#pragma once
class Calculator
{
public:
    double Calculate(double x, char oper, double y);
};
```

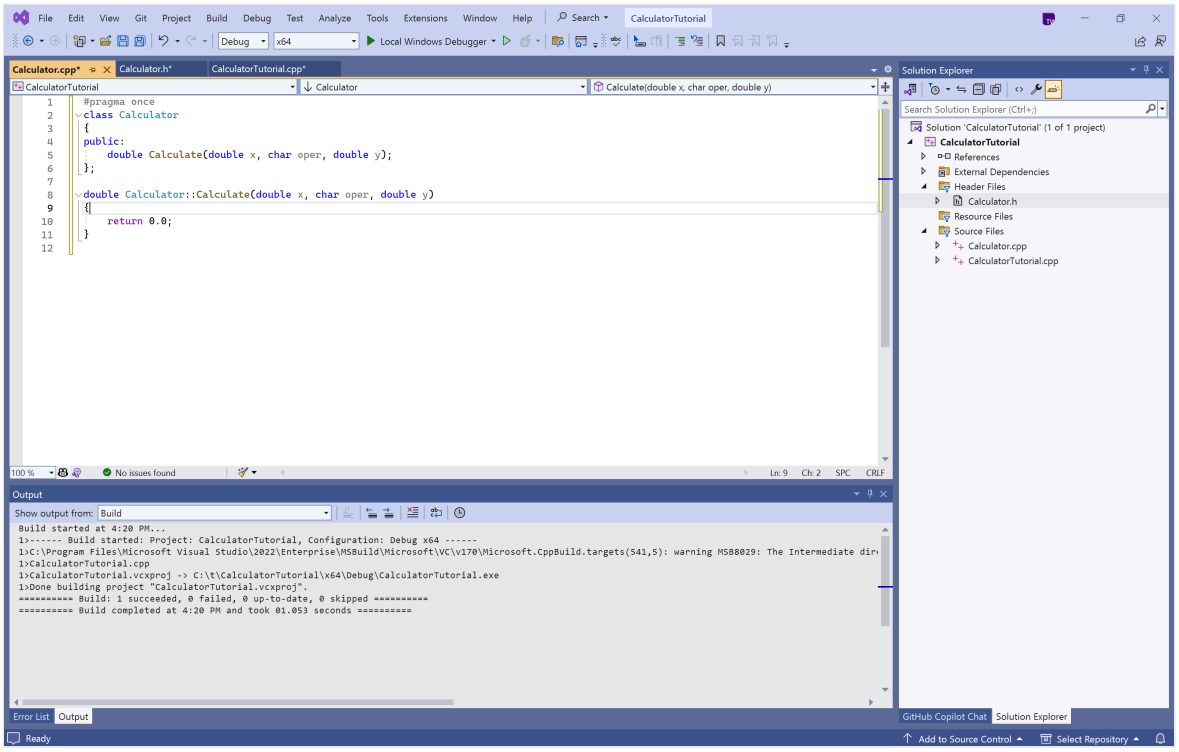
> Understanding the code
>
> - This code declares a new function called `Calculate`, which handles math operations for addition, subtraction, multiplication, and division.

- C++ code is organized into *header* (`.h`) files and *source* (`.cpp`) files. Some other file extensions are supported by various compilers, but these are the main ones to know about. Functions and variables are normally *declared*, that is, given a name and a type, in header files, and *implemented*, or given a definition, in source files. To access code defined in another file, you can use `#include "filename.h"`, where `filename.h` is the name of the file that declares the variables or functions you want to use.
- It's good practice to organize your code into different files based on what it does, so it's easy to find the code you need later. In our case, we define the `Calculator` class separately from the file containing the `main()` function, but we plan to reference the `Calculator` class in `main()`.

3. A green squiggle appears under `Calculate` because although the `Calculate` function is *declared*, it isn't *defined*. Hover over `Calculate`, click the down arrow on the screwdriver icon, and select **Create definition of 'Calculate' in `Calculator.cpp`**.



This code is added to `Calculator.cpp`:



Currently, it just returns 0.0. Let's change that.

4. Switch to the `Calculator.cpp` file in the editor window. Replace the contents of `Calculator::Calculate(double x, char oper, double y)` with:

C++

```
double Calculator::Calculate(double x, char oper, double y)
{
    switch(oper)
    {
        case '+':
            return x + y;
        case '-':
            return x - y;
        case '*':
            return x * y;
        case '/':
            return x / y;
        default:
            return 0.0;
    }
}
```

> Understanding the code
>
> - The function `Calculate` takes a number, an operator, and a second number. Then it performs the requested operation on the two numbers.
> - The `switch` statement checks which operator was provided, and executes the case corresponding to that operation. The `default:` case is a fallback

in case the user types an operator that isn't handled by any of the preceding `case` statements. It's best to handle invalid user input in a more elegant way, but this is beyond the scope of this tutorial.

- The **`double`** keyword denotes a type of number that supports decimals. This type of number is called a floating-point number, and `double` means a floating point number that has extra precision. This way, the calculator can handle both decimal math and integer math. The `Calculate` function is required to always return a double-precision floating point number due to the **`double`** at the start of the code (this denotes the function's return type), which is why we return 0.0 in the default case.
- The `.h` file declares the function *prototype*, which tells the compiler upfront what parameters it requires, and what return type to expect from it. The `.cpp` file has all the implementation details of the function.

If you build and run the code again at this point, it immediately exits after asking which operation to perform. So, modify the `main` function to do multiple calculations.

Call the `Calculator` class member functions

1. Update the `main` function in *`CalculatorTutorial.cpp`* as follows:

C++

```
// CalculatorTutorial.cpp : This file contains the 'main' function.
Program execution begins and ends there.
//

#include <iostream>
#include "Calculator.h"

using namespace std;

int main()
{
    double x = 0.0;
    double y = 0.0;
    double result = 0.0;
    char oper = '+';

    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b |
a*b | a/b"
         << endl;

    Calculator c;
    while (true)
    {
```

```
        cin >> x >> oper >> y;
        result = c.Calculate(x, oper, y);
        cout << "Result " << "of " << x << oper << y << " is: " <<
result << endl;
    }

    return 0;
}
```

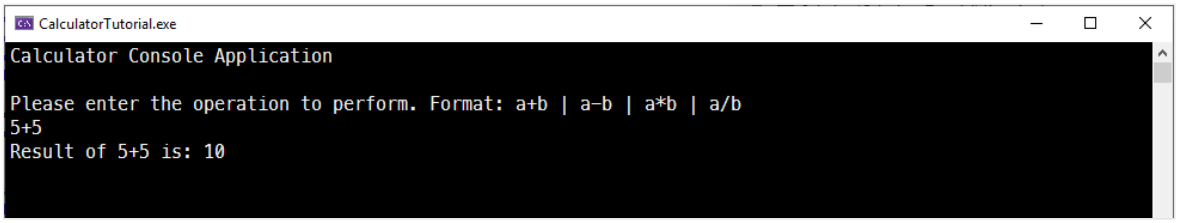
> Understanding the code
>
> - Since C++ programs always start at the `main()` function, we need to call our other code from there, so an `#include` statement is needed to make that code visible to our `main()` function.
> - The variables `x`, `y`, `oper`, and `result` are declared to store the first number, second number, operator, and final result, respectively. It's always good practice to give them some initial values to avoid undefined behavior, which is what is done here.
> - The `Calculator c;` line declares an object named `c` as an instance of the `Calculator` class. The class itself is just a blueprint for how calculators work; the object is the specific calculator that does the math.
> - The `while (true)` statement is a loop. The code inside the loop executes over and over again as long as the condition inside the `()` holds true. Since the condition is simply listed as **`true`**, it's always true, so the loop runs forever. To close the program, the user must manually close the console window. Otherwise, the program always waits for new input.
> - The `cin` keyword accepts input from the user. The input stream is smart enough to process a line of text entered in the console window and place it inside each of the variables listed, in order.
> - The `c.Calculate(x, oper, y);` expression calls the `Calculate` function defined earlier, and supplies the entered input values and the requested operation. The function then returns a number that is stored in `result`.
> - Finally, `result` is printed to the console and the user sees the result of the calculation.

Build and test the code again

Now test the program again to make sure everything works properly.

1. Press **Ctrl+F5** to rebuild and start the app.

2. Enter `5+5`, and press **Enter**. Verify that the result is 10.



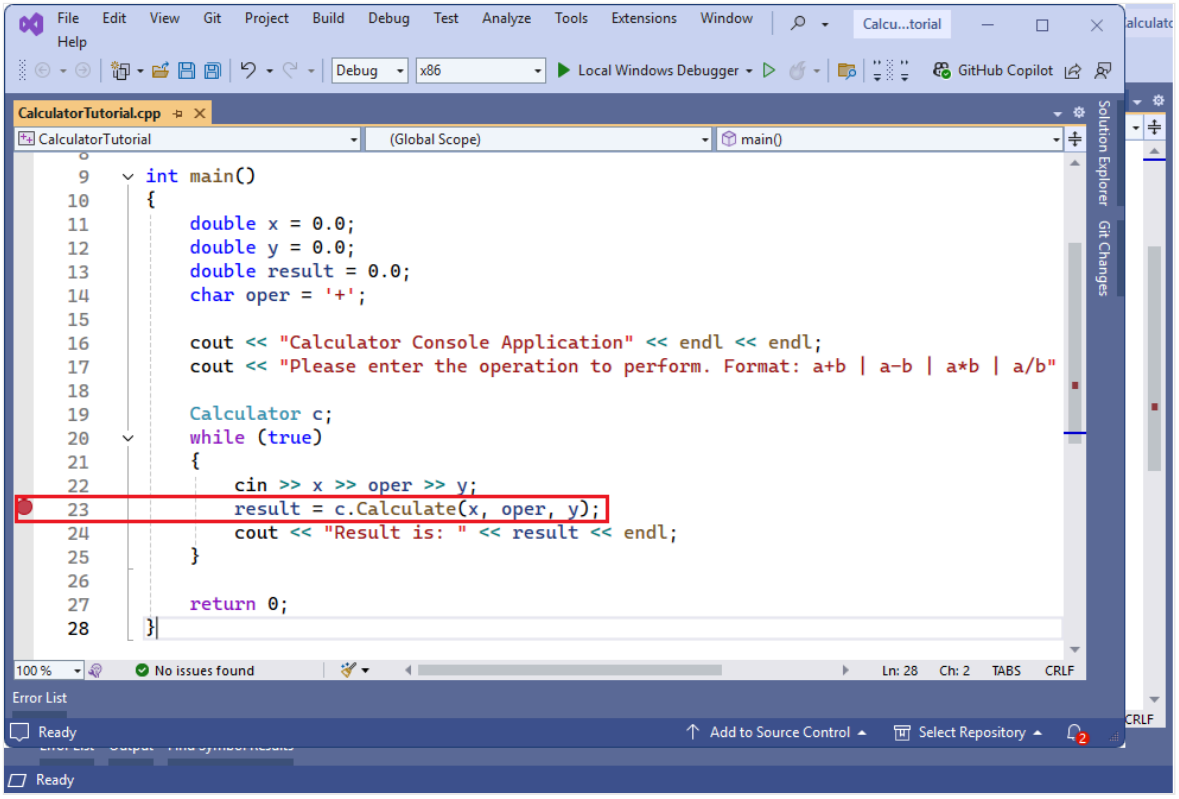
3. Stop the program by closing the console window.

Debug the app

Since the user is free to type anything into the console window, let's make sure the calculator handles unexpected input. Instead of running the program, let's debug it so we can inspect what it's doing step-by-step.

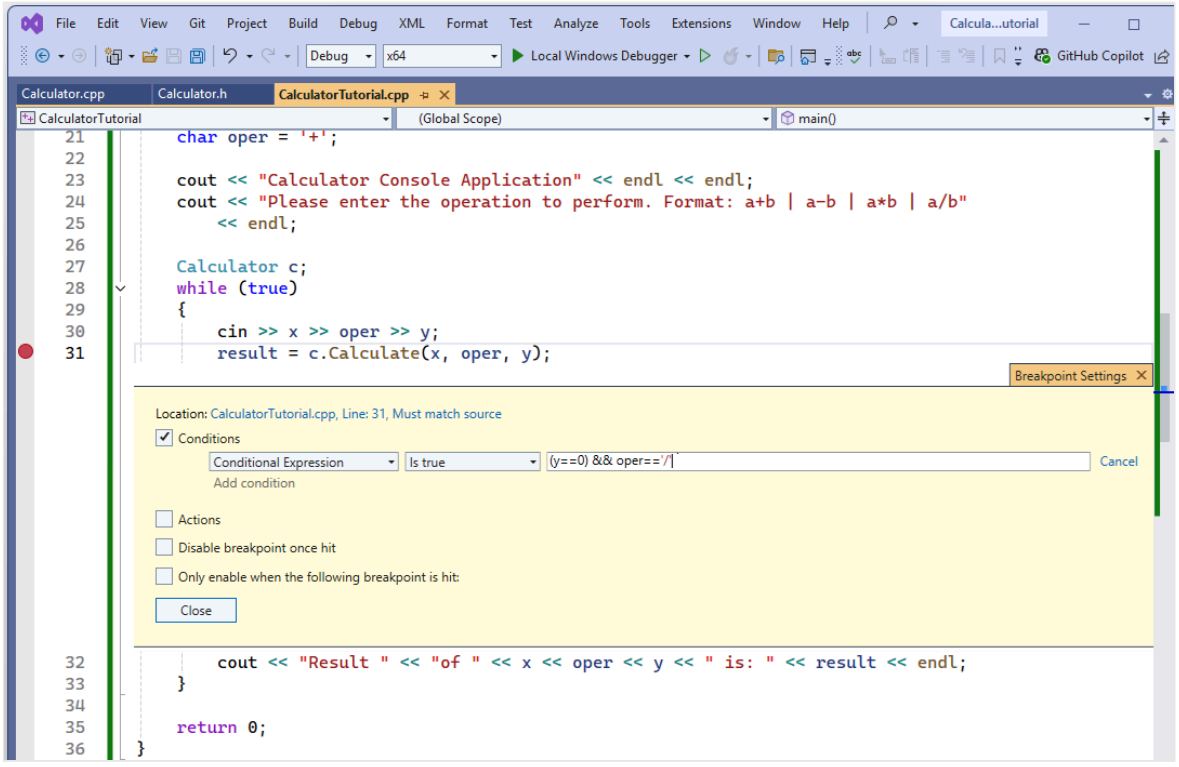
Run the app in the debugger

1. In `CalcuatorTutorial.cpp`, set a breakpoint on the line: `result = c.Calculate(x, oper, y);`. To set the breakpoint, click next to the line in the gray vertical bar along the left edge of the editor window so that a red dot appears.



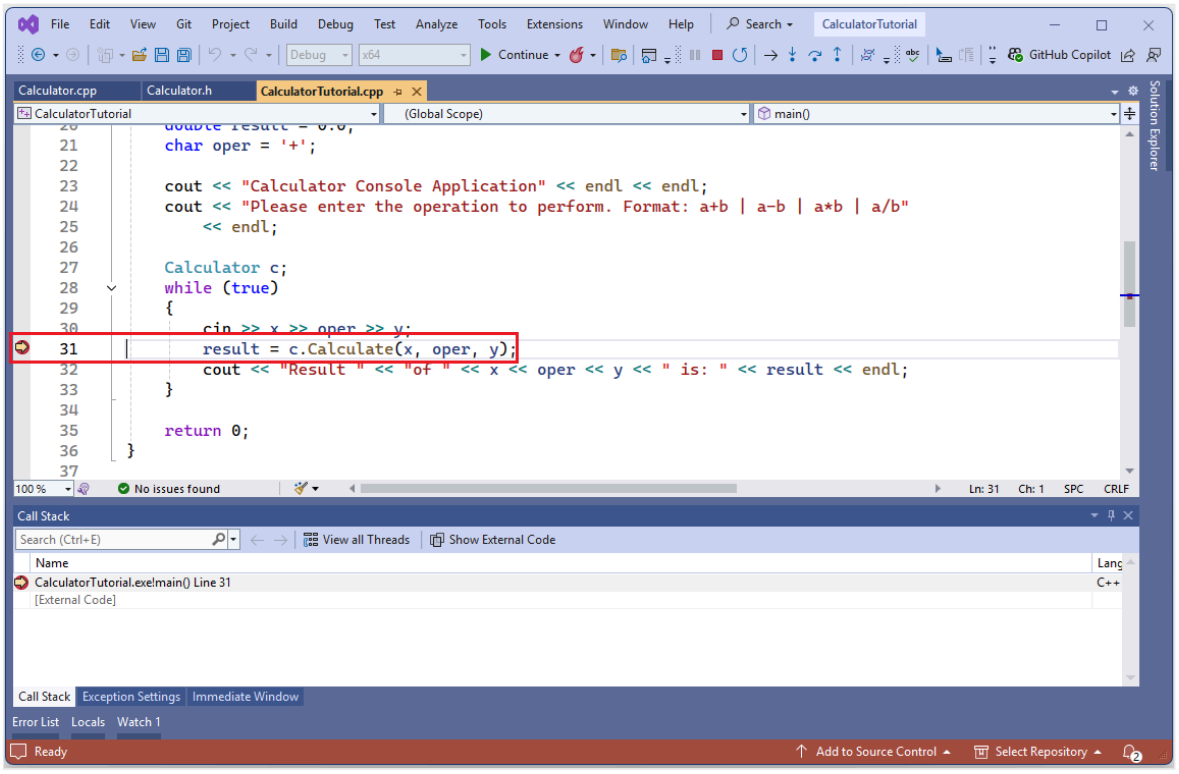
Now when we debug the program, execution pauses at that line. We already have a rough idea that the program works for simple cases. Since we don't want to pause execution every time we call `Calculate()`, let's make the breakpoint conditional.

2. Right-click the red dot that represents the breakpoint, and select **Conditions**. In the edit box for the condition, enter `(y == 0) && (oper == '/')`. Select the **Close** button to save the breakpoint condition.



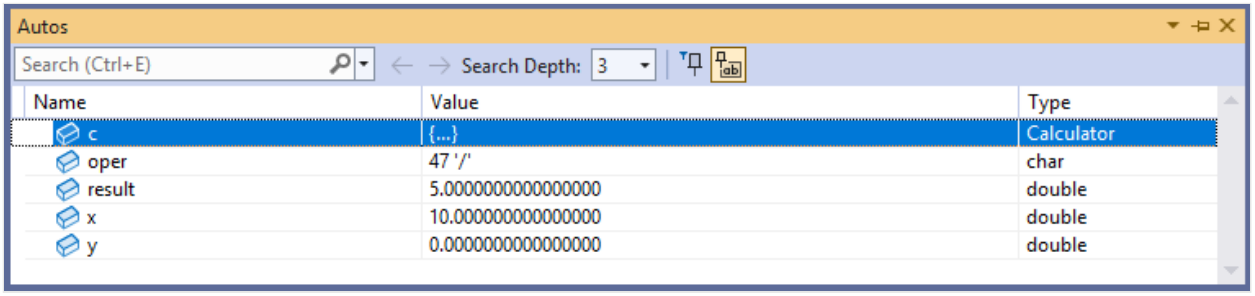
Now, execution pauses at the breakpoint when the app tries to divide by 0.

3. To debug the program, press **F5**, or select the **Local Windows Debugger** debugger toolbar button that has the green arrow icon. In your console app, if you enter something like "5 - 0", the program behaves normally and keeps running. However, if you type "10 / 0", it pauses at the breakpoint. You can put any number of spaces between the operator and numbers: `cin` is smart enough to parse the input appropriately.

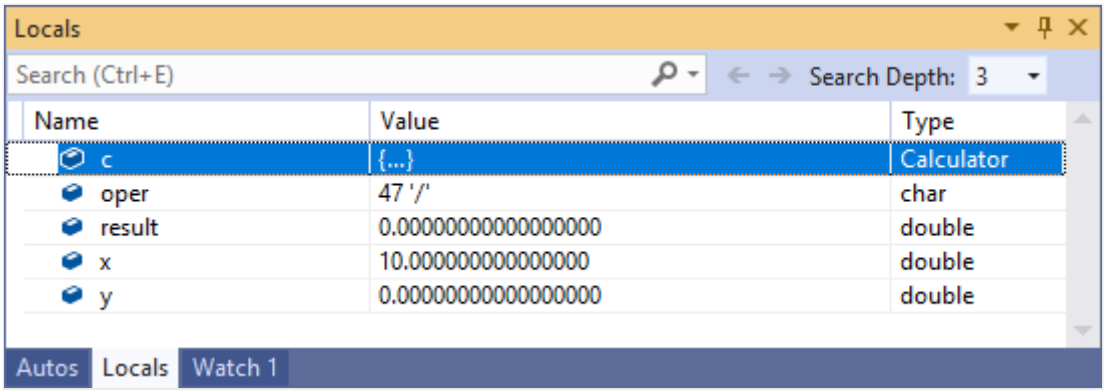


Useful windows in the debugger

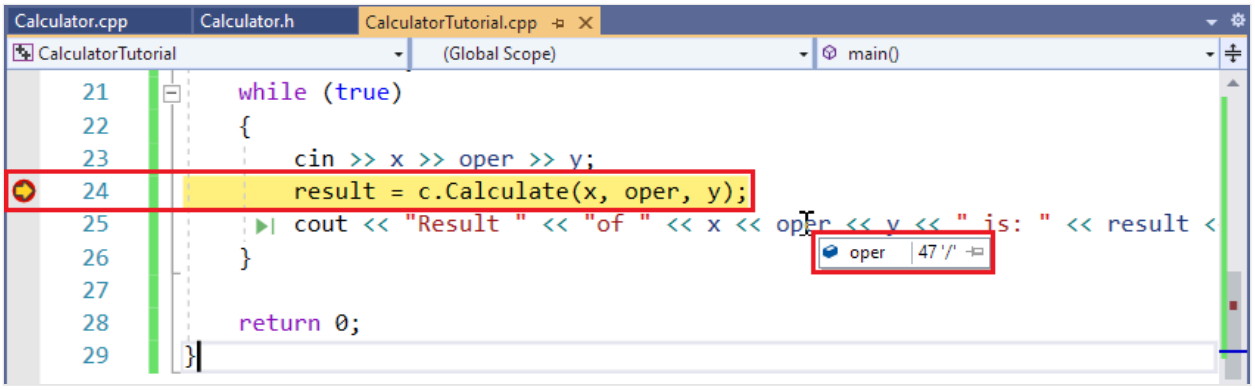
When you debug your code, you may notice that some new windows appear. These windows can assist your debugging experience. Take a look at the **Autos** window. The **Autos** window shows you the current values of variables used at least three lines before and up to the current line. If you don't see the **Autos** window, from the main menu select **Debug** > **Windows** > **Autos**.



To see all of the variables from that function, switch to the **Locals** window. Because this is a small function, the Autos and Locals window show the same variables. But you can modify the values of these variables in the Locals window while debugging to see what effect they would have on the program. In this case, we leave them alone. Open the **Locals** window by selecting **Locals** at the bottom of the **Autos** window, or by selecting from the main menu **Debug** > **Windows** > **Locals**.



You can also hover over variables in the code to see their current values at the point where execution is currently paused. Make sure the editor window is in focus by clicking on it first.



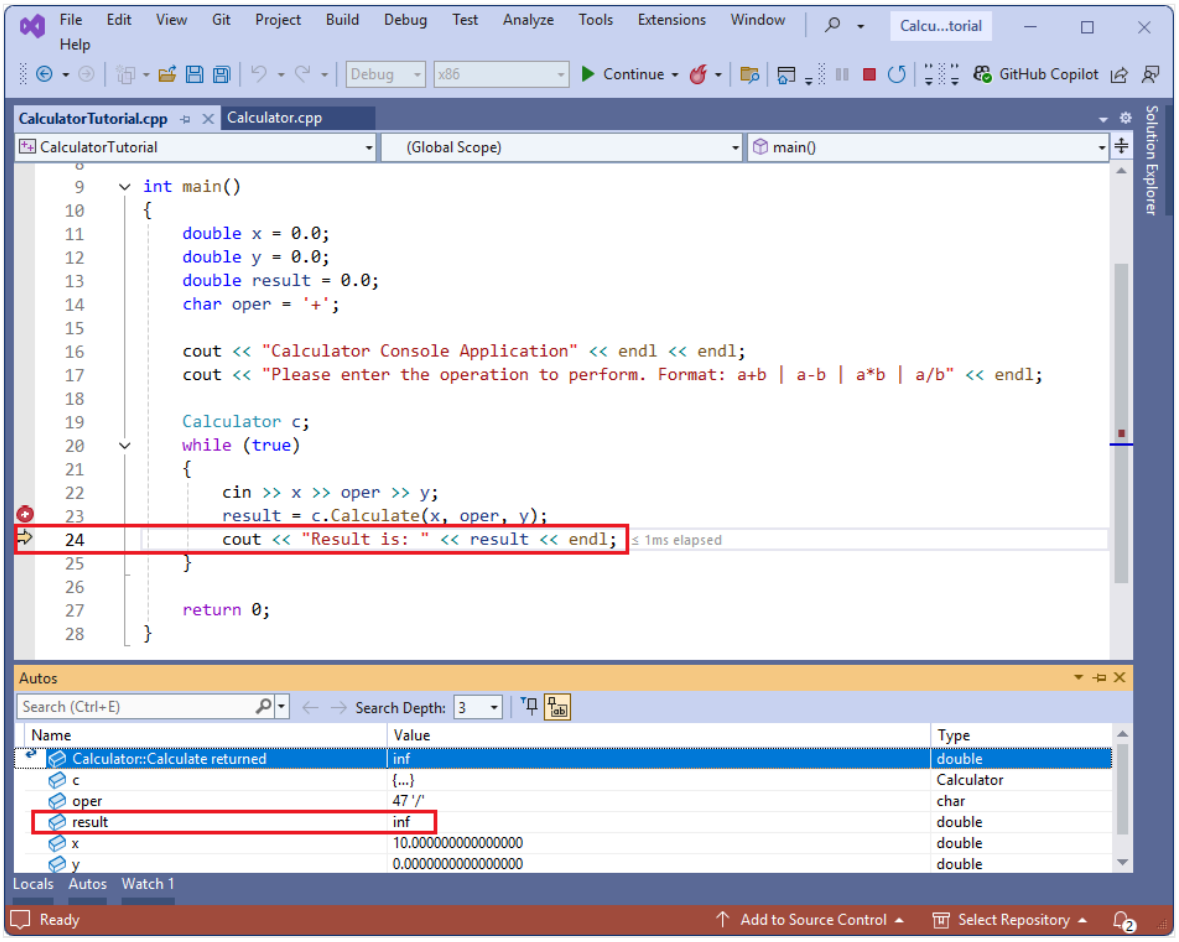
Continue debugging

1. The yellow arrow on the left shows the current point of execution. The current line calls `Calculate`, so press **F11** to **Step Into** the function. Now you're executing code in the body of the `Calculate` function. Be careful with **Step Into** because it steps into any functions on the line you're on, including standard library functions. It's fine to step into the standard library, but you may be more interested in focusing on your code instead of library code.

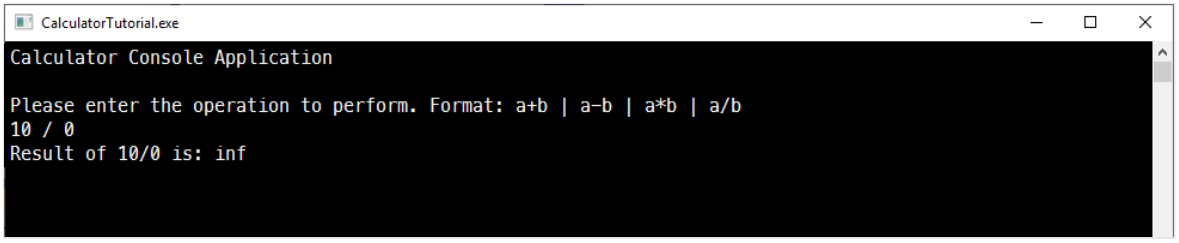
2. Now that the point of execution is at the start of the `Calculate` function, press **F10** to move to the next line in the program's execution. **F10** is also known as **Step Over**. You can use **Step Over** to move from line to line, without delving into the details of what is occurring in each part of the line. In general, you should use **Step Over** instead of **Step Into** unless you want to dive more deeply into code that is being called from elsewhere (as you did to reach the body of `Calculate`).

3. Continue using **F10** to **Step Over** each line until you get back to the `main()` function in the other file, and stop on the `cout` line.

The program is doing what's expected: it takes the first number, and divides it by the second. On the `cout` line, hover over the `result` variable or take a look at `result` in the **Autos** window. Its value is `inf`, which doesn't look right.



Let's fix it. The `cout` line outputs whatever value is stored in `result`, so when you step one more line forward using **F10**, the console window displays:



This result is because division by zero is undefined, so the program doesn't have a numerical answer for the requested operation.

Fix the "divide by zero" error

Let's handle division by zero more gracefully so that it's easier for the user to understand the problem.

1. Make the following changes in *`CalculatorTutorial.cpp`*. You can leave the program running as you edit, thanks to a debugger feature called **Edit and**

Continue. Add an `if` statement following `cin >> x >> oper >> y;` to check for division by zero and output a message to the user if it happens. Otherwise, the result is printed.

C++

```
// CalculatorTutorial.cpp : This file contains the 'main' function.
Program execution begins and ends there.
//

#include <iostream>
#include "Calculator.h"

using namespace std;

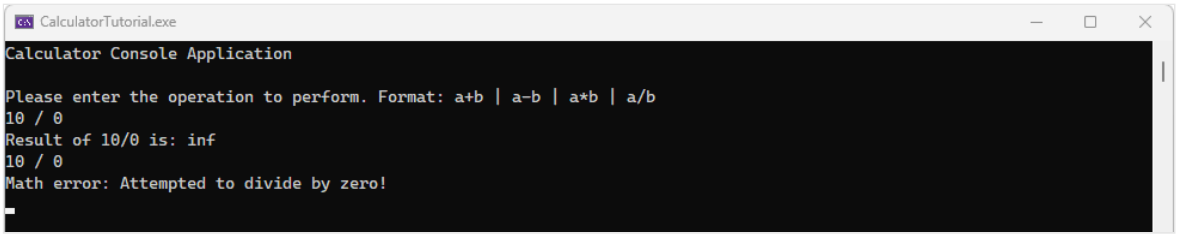
int main()
{
    double x = 0.0;
    double y = 0.0;
    double result = 0.0;
    char oper = '+';

    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b |
a*b | a/b" << endl;

    Calculator c;
    while (true)
    {
        cin >> x >> oper >> y;
        if (oper == '/' && y == 0)
        {
            cout << "Math error: Attempted to divide by zero!" << endl;
            continue;
        }
        else
        {
            result = c.Calculate(x, oper, y);
        }
        cout << "Result " << "of " << x << oper << y << " is: " <<
result << endl;
    }

    return 0;
}
```

2. Press **F5** once. Program execution continues until it has to pause to ask for user input. Enter `10 / 0` again. Now, a more helpful message is printed. The user is asked for more input, and the program continues executing normally.



> **Note**
>
> When you edit code while in debugging mode, there's a risk of code becoming stale. This happens when the debugger is still running your old code, and has not yet updated it with your changes. The debugger displays a dialog to inform you when this happens. Sometimes, you may need to press **F5** to refresh the code being executed. In particular, if you make a change inside a function while the point of execution is inside that function, you need to step out of the function, then back into it again to get the updated code. If that doesn't work and you see an error message, you can stop debugging by clicking on the red square in the toolbar under the menus at the top of the IDE, then start debugging again by entering **F5** or by choosing the green "play" arrow beside the stop button on the toolbar.
>
> Another reason edit and continue may fail is if you see a message that says "The Require source files to exactly match the original version setting under Debug->Options->General needs to be enabled..." To fix this, from the main menu select **Tools** > **Options** > **Debugging** > **General** and ensure that **Require source files to exactly match the original version** is checked.

> Understanding the Run and Debug shortcuts
>
> - **F5**, or **Debug** > **Start Debugging**, starts a debugging session, if one isn't already active, and runs the program until a breakpoint is hit or the program needs user input. If no user input is needed and no breakpoint is available to hit, the program terminates and the console window closes itself when the program finishes running. If your program outputs to the console, use **Ctrl+F5** or set a breakpoint before you press **F5** to keep the window open.
> - **Ctrl+F5**, or **Debug** > **Start Without Debugging**, runs the application without going into debug mode. This is slightly faster than debugging, and the console window stays open after the program finishes executing.
> - **F10**, known as **Step Over**, lets you iterate through code, line-by-line, and visualize how the code is run and what variable values are at each step of execution.

> - **F11**, known as **Step Into**, works similarly to **Step Over**, except it steps into any functions called on the line of execution. For example, if the line being executed calls a function, pressing **F11** moves the pointer into the body of the function, so you can follow the function's code being run before coming back to the line you started at. Pressing **F10** steps over the function call and just moves to the next line; the function call still happens, but the program doesn't pause to show you what it's doing.

Close the app

- If it's still running, close the console window to stop the calculator app.

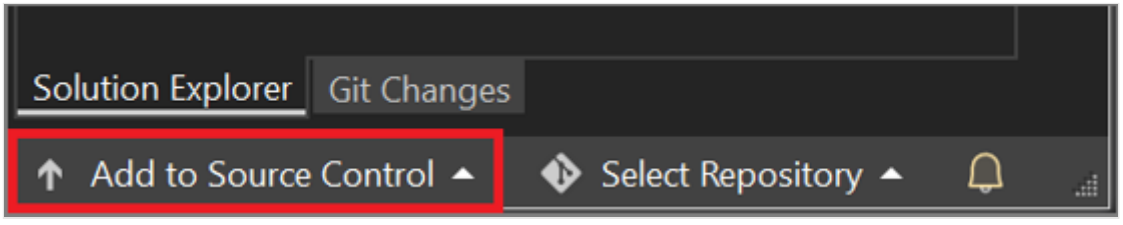
Add Git source control

Now that you've created an app, you might want to add it to a Git repository. We've got you covered. Visual Studio makes that process easy with Git tools you can use directly from the IDE.

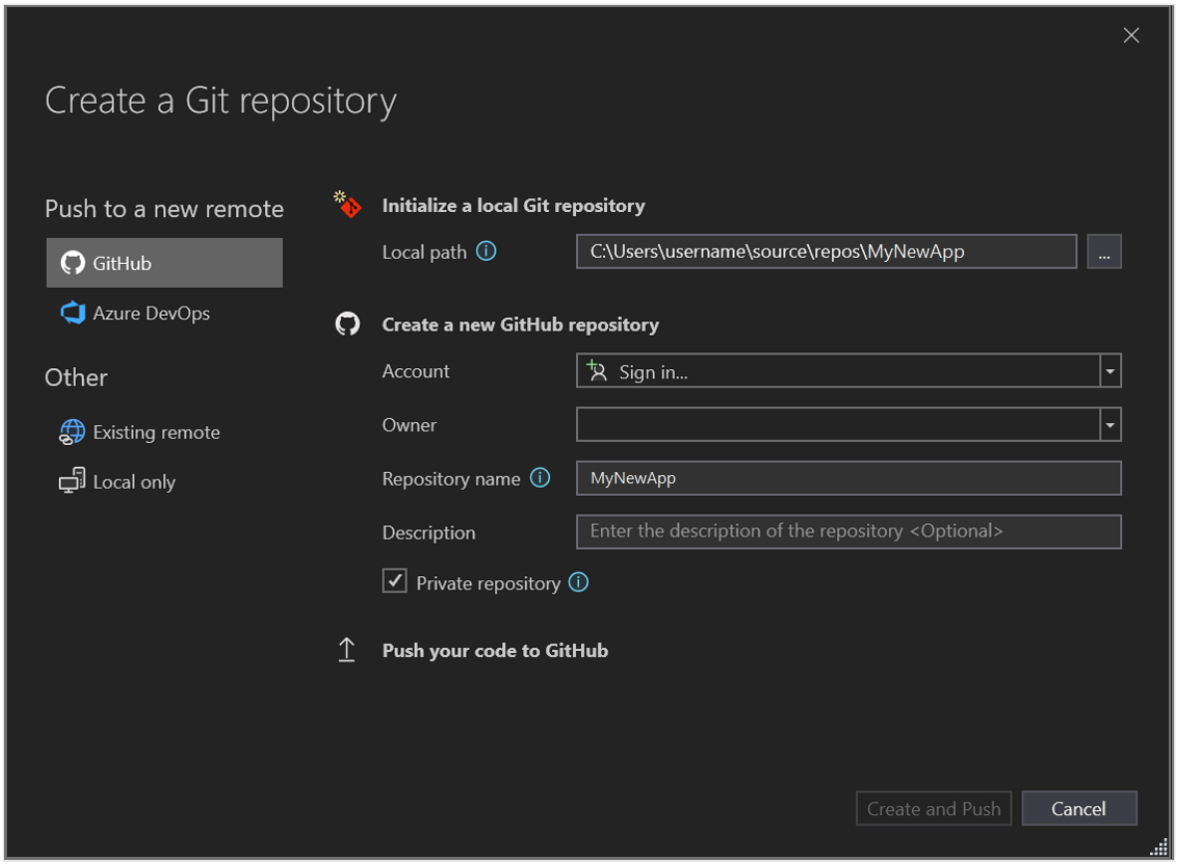
> **Tip**
>
> Git is the most widely used modern version control system, so whether you're a professional developer or you're learning how to code, Git can be very useful. If you're new to Git, the https://git-scm.com/ website is a good place to start. There, you can find cheat sheets, a popular online book, and Git Basics videos.

To associate your code with Git, you start by creating a new Git repository where your code is located. Here's how:

1. In the status bar at the bottom-right corner of Visual Studio, select **Add to Source Control**, and then select **Git**.



2. In the **Create a Git repository** dialog box, sign in to GitHub.

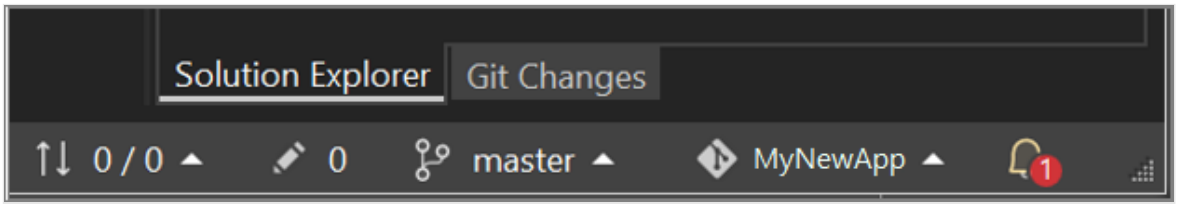


The repository name auto-populates based on your folder location. By default, your new repository is private, which means you're the only one who can access it.

> **Tip**
>
> Whether your repository is public or private, it's best to have a remote backup of your code stored securely on GitHub. Even if you aren't working with a team, a remote repository makes your code available to you from any computer.

3. Select **Create and Push**.

After you create your repository, status details appear in the status bar.



The first icon with the arrows shows how many outgoing/incoming commits are in your current branch. You can use this icon to pull any incoming commits or push any outgoing commits. You can also choose to view these commits first. To do so, select the icon, and then select **View Outgoing/Incoming**.

The second icon with the pencil shows the number of uncommitted changes to your code. You can select this icon to view those changes in the **Git Changes** window.

To learn more about how to use Git with your app, see the Visual Studio version control documentation.

The finished app

Congratulations! You completed the code for the calculator app, built and debugged it, and added it to a repo, all in Visual Studio.

Next steps

Learn more about Visual Studio for C++

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Get started with C++/WinRT

Article • 02/13/2023

> **Important**
>
> For info about setting up Visual Studio for C++/WinRT development—including installing and using the C++/WinRT Visual Studio Extension (VSIX) and the NuGet package (which together provide project template and build support)—see **Visual Studio support for C++/WinRT**.

To get you up to speed with using C++/WinRT, this topic walks through a simple code example based on a new **Windows Console Application (C++/WinRT)** project. This topic also shows how to add C++/WinRT support to a Windows Desktop application project.

> **Note**
>
> While we recommend that you develop with the latest versions of Visual Studio and the Windows SDK, if you're using Visual Studio 2017 (version 15.8.0 or later), and targeting the Windows SDK version 10.0.17134.0 (Windows 10, version 1803), then a newly created C++/WinRT project may fail to compile with the error "*error C3861: 'from_abi': identifier not found*", and with other errors originating in *base.h*. The solution is to either target a later (more conformant) version of the Windows SDK, or set project property **C/C++** > **Language** > **Conformance mode: No** (also, if **/permissive-** appears in project property **C/C++** > **Language** > **Command Line** under **Additional Options**, then delete it).

A C++/WinRT quick-start

Create a new **Windows Console Application (C++/WinRT)** project.

Edit `pch.h` and `main.cpp` to look like this.

C++/WinRT

```
// pch.h
#pragma once
#include <winrt/Windows.Foundation.Collections.h>
```

```
#include <winrt/Windows.Web.Syndication.h>
#include <iostream>
```

C++/WinRT

```
// main.cpp
#include "pch.h"

using namespace winrt;
using namespace Windows::Foundation;
using namespace Windows::Web::Syndication;

int main()
{
    winrt::init_apartment();

    Uri rssFeedUri{ L"https://blogs.windows.com/feed" };
    SyndicationClient syndicationClient;
    syndicationClient.SetRequestHeader(L"User-Agent", L"Mozilla/5.0
(compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)");
    SyndicationFeed syndicationFeed =
syndicationClient.RetrieveFeedAsync(rssFeedUri).get();
    for (const SyndicationItem syndicationItem : syndicationFeed.Items())
    {
        winrt::hstring titleAsHstring = syndicationItem.Title().Text();

        // A workaround to remove the trademark symbol from the title
string, because it causes issues in this case.
        std::wstring titleAsStdWstring{ titleAsHstring.c_str() };
        titleAsStdWstring.erase(remove(titleAsStdWstring.begin(),
titleAsStdWstring.end(), L'™'), titleAsStdWstring.end());
        titleAsHstring = titleAsStdWstring;

        std::wcout << titleAsHstring.c_str() << std::endl;
    }
}
```

Let's take the short code example above piece by piece, and explain what's going on in each part.

C++/WinRT

```
#include <winrt/Windows.Foundation.Collections.h>
#include <winrt/Windows.Web.Syndication.h>
```

With the default project settings, the included headers come from the Windows SDK, inside the folder `%WindowsSdkDir%Include<WindowsTargetPlatformVersion>\cppwinrt\winrt`. Visual Studio includes that path in its *IncludePath* macro. But there's no strict dependency on the

Windows SDK, because your project (via the `cppwinrt.exe` tool) generates those same headers into your project's *$(GeneratedFilesDir)* folder. They'll be loaded from that folder if they can't be found elsewhere, or if you change your project settings.

The headers contain Windows APIs projected into C++/WinRT. In other words, for each Windows type, C++/WinRT defines a C++-friendly equivalent (called the *projected type*). A projected type has the same fully-qualified name as the Windows type, but it's placed in the C++ **winrt** namespace. Putting these includes in your precompiled header reduces incremental build times.

> **ⓘ Important**
>
> Whenever you want to use a type from a Windows namespaces, you must `#include` the corresponding C++/WinRT Windows namespace header file, as shown above. The *corresponding* header is the one with the same name as the type's namespace. For example, to use the C++/WinRT projection for the Windows::Foundation::Collections::PropertySet runtime class, include the `winrt/Windows.Foundation.Collections.h` header.
>
> It is common for a C++/WinRT projection header to automatically include related namespace header files. For example, `winrt/Windows.Foundation.Collections.h` includes `winrt/Windows.Foundation.h`. But you shouldn't rely on this behavior, since it's an implementation detail that changes over time. You must explicitly include any headers that you need.

C++/WinRT

```
using namespace winrt;
using namespace Windows::Foundation;
using namespace Windows::Web::Syndication;
```

The `using namespace` directives are optional, but convenient. The pattern shown above for such directives (allowing unqualified name lookup for anything in the **winrt** namespace) is suitable for when you're beginning a new project and C++/WinRT is the only language projection you're using inside of that project. If, on the other hand, you're mixing C++/WinRT code with C++/CX and/or SDK application binary interface (ABI) code (you're either porting from, or interoperating with, one or both of those models), then see the topics Interop between C++/WinRT and C++/CX, Move to C++/WinRT from C++/CX, and Interop between C++/WinRT and the ABI.

C++/WinRT

```
winrt::init_apartment();
```

The call to **winrt::init_apartment** initializes the thread in the Windows Runtime; by default, in a multithreaded apartment. The call also initializes COM.

C++/WinRT

```
Uri rssFeedUri{ L"https://blogs.windows.com/feed" };
SyndicationClient syndicationClient;
```

Stack-allocate two objects: they represent the uri of the Windows blog, and a syndication client. We construct the uri with a simple wide string literal (see String handling in C++/WinRT for more ways you can work with strings).

C++/WinRT

```
SyndicationFeed syndicationFeed =
syndicationClient.RetrieveFeedAsync(rssFeedUri).get();
```

SyndicationClient::RetrieveFeedAsync is an example of an asynchronous Windows Runtime function. The code example receives an asynchronous operation object from **RetrieveFeedAsync**, and it calls **get** on that object to block the calling thread and wait for the result (which is a syndication feed, in this case). For more about concurrency, and for non-blocking techniques, see Concurrency and asynchronous operations with C++/WinRT.

C++/WinRT

```
for (const SyndicationItem syndicationItem : syndicationFeed.Items()) { ...
}
```

SyndicationFeed.Items is a range, defined by the iterators returned from **begin** and **end** functions (or their constant, reverse, and constant-reverse variants). Because of this, you can enumerate **Items** with either a range-based `for` statement, or with the **std::for_each** template function. Whenever you iterate over a Windows Runtime collection like this, you'll need to `#include <winrt/Windows.Foundation.Collections.h>`.

C++/WinRT

```
winrt::hstring titleAsHstring = syndicationItem.Title().Text();

// Omitted: there's a little bit of extra work here to remove the trademark
symbol from the title text.
```

```
std::wcout << titleAsHstring.c_str() << std::endl;
```

Gets the feed's title text, as a **winrt::hstring** object (more details in String handling in C++/WinRT). The **hstring** is then output, via the **c_str** function, which reflects the pattern used with C++ Standard Library strings.

As you can see, C++/WinRT encourages modern, and class-like, C++ expressions such as `syndicationItem.Title().Text()`. This is a different, and cleaner, programming style from traditional COM programming. You don't need to directly initialize COM, nor work with COM pointers.

Nor do you need to handle HRESULT return codes. C++/WinRT converts error HRESULTs to exceptions such as **winrt::hresult-error** for a natural and modern programming style. For more info about error-handling, and code examples, see Error handling with C++/WinRT.

Modify a Windows Desktop application project to add C++/WinRT support

Some desktop projects (for example, the WinUI 3 templates in Visual Studio) have C++/WinRT support built in.

But this section shows you how you can add C++/WinRT support to any Windows Desktop application project that you might have. If you don't have an existing Windows Desktop application project, then you can follow along with these steps by first creating one. For example, open Visual Studio and create a **Visual C++** > **Windows Desktop** > **Windows Desktop Application** project.

You can optionally install the C++/WinRT Visual Studio Extension (VSIX) and the NuGet package. For details, see Visual Studio support for C++/WinRT.

Set project properties

Go to project property **General** > **Windows SDK Version**, and select **All Configurations** and **All Platforms**. Ensure that **Windows SDK Version** is set to 10.0.17134.0 (Windows 10, version 1803) or greater.

Confirm that you're not affected by Why won't my new project compile?.

Because C++/WinRT uses features from the C++17 standard, set project property **C/C++** > **Language** > **C++ Language Standard** to *ISO C++17 Standard (/std:c++17)*.

The precompiled header

The default project template creates a precompiled header for you, named either `framework.h`, or `stdafx.h`. Rename that to `pch.h`. If you have a `stdafx.cpp` file, then rename that to `pch.cpp`. Set project property **C/C++** > **Precompiled Headers** > **Precompiled Header** to *Create (/Yc)*, and **Precompiled Header File** to *pch.h*.

Find and replace all `#include "framework.h"` (or `#include "stdafx.h"`) with `#include "pch.h"`.

In `pch.h`, include `winrt/base.h`.

C++/WinRT

```
// pch.h
...
#include <winrt/base.h>
```

Linking

The C++/WinRT language projection depends on certain Windows Runtime free (non-member) functions, and entry points, that require linking to the WindowsApp.lib umbrella library. This section describes three ways of satisfying the linker.

The first option is to add to your Visual Studio project all of the C++/WinRT MSBuild properties and targets. To do this, install the Microsoft.Windows.CppWinRT NuGet package into your project. Open the project in Visual Studio, click **Project** > **Manage NuGet Packages...** > **Browse**, type or paste **Microsoft.Windows.CppWinRT** in the search box, select the item in search results, and then click **Install** to install the package for that project.

You can also use project link settings to explicitly link `WindowsApp.lib`. Or, you can do it in source code (in `pch.h`, for example) like this.

C++/WinRT

```
#pragma comment(lib, "windowsapp")
```

You can now compile and link, and add C++/WinRT code to your project (for example, code similar to that shown in the A C++/WinRT quick-start section, above).

The three main scenarios for C++/WinRT

As you use and become familiar with C++/WinRT, and work through the rest of the documentation here, you'll likely notice that there are three main scenarios, as described in the following sections.

Consuming Windows APIs and types

In other words, *using*, or *calling* APIs. For example, making API calls to communicate using Bluetooth; to stream and present video; to integrate with the Windows shell; and so on. C++/WinRT fully and uncompromisingly supports this category of scenario. For more info, see Consume APIs with C++/WinRT.

Authoring Windows APIs and types

In other words, *producing* APIs and types. For example, producing the kinds of APIs described in the section above; or the graphics APIs; the storage and file system APIs; the networking APIs, and so on. For more info, see Author APIs with C++/WinRT.

Authoring APIs with C++/WinRT is a little more involved than consuming them, because you must use IDL to define the shape of the API before you can implement it. There's a walkthrough of doing that in XAML controls; bind to a C++/WinRT property.

XAML applications

This scenario is about building applications and controls on the XAML UI framework. Working in a XAML application amounts to a combination of consuming and authoring. But since XAML is the dominant UI framework on Windows today, and its influence over the Windows Runtime is proportionate to that, it deserves its own category of scenario.

Be aware that XAML works best with programming languages that offer reflection. In C++/WinRT, you sometimes have to do a little extra work in order to interoperate with the XAML framework. All of those cases are covered in the documentation. Good places to start are XAML controls; bind to a C++/WinRT property and XAML custom (templated) controls with C++/WinRT.

Sample apps written in C++/WinRT

See Where can I find C++/WinRT sample apps?.

Important APIs

- SyndicationClient::RetrieveFeedAsync method
- SyndicationFeed.Items property
- winrt::hstring struct
- winrt::hresult-error struct

Related topics

- C++/CX
- Error handling with C++/WinRT
- Interop between C++/WinRT and C++/CX
- Interop between C++/WinRT and the ABI
- Move to C++/WinRT from C++/CX
- String handling in C++/WinRT

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Get Started with Win32 and C++

Article • 01/27/2022

The aim of this Get Started series is to teach you how to write a desktop program in C++ using Win32 and COM APIs.

In the first module, you'll learn step-by-step how to create and show a window. Later modules will introduce the Component Object Model (COM), graphics and text, and user input.

For this series, it is assumed that you have a good working knowledge of C++ programming. No previous experience with Windows programming is assumed. If you are new to C++, learning material is available in the C++ language documentation.

In this section

Topic	Description
Intro to Win32 programming in C++	This section describes some of the basic terminology and coding conventions used in Windows programming.
Module 1. Your First Windows Program	In this module, you will create a simple Windows program that shows a blank window.
Module 2. Using COM in Your Windows Program	This module introduces the Component Object Model (COM), which underlies many of the modern Windows APIs.
Module 3. Windows Graphics	This module introduces the Windows graphics architecture, with a focus on Direct2D.
Module 4. User Input	This module describes mouse and keyboard input.
Sample Code	Contains links to download the sample code for this series.

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

Create a simple Universal Windows Platform (UWP) game with DirectX

Article • 10/20/2022

In this set of tutorials, you'll learn how to use DirectX and C++/WinRT to create the basic Universal Windows Platform (UWP) sample game named **Simple3DGameDX**. The gameplay takes place in a simple first-person 3D shooting gallery.

> **Note**
>
> The link from which you can download the **Simple3DGameDX** sample game itself is **Direct3D sample game**. The C++/WinRT source code is in the folder named `cppwinrt`. For info about other UWP sample apps, see **Sample applications for Windows development**.

These tutorials cover all of the major parts of a game, including the processes for loading assets such as arts and meshes, creating a main game loop, implementing a simple rendering pipeline, and adding sound and controls.

You'll also see UWP game development techniques and considerations. We'll focus on key UWP DirectX game development concepts, and call out Windows-Runtime-specific considerations around those concepts.

Objective

To learn about the basic concepts and components of a UWP DirectX game, and to become more comfortable designing UWP games with DirectX.

What you need to know

For this tutorial, you need to be familiar with these subjects.

- C++/WinRT. C++/WinRT is a standard modern C++17 language projection for Windows APIs, implemented as a header-file-based library, and designed to provide you with first-class access to the modern Windows APIs.
- Basic linear algebra and Newtonian physics concepts.
- Basic graphics programming terminology.
- Basic Windows programming concepts.
- Basic familiarity with the Direct2D and Direct3D 11 APIs.

Direct3D UWP shooting gallery sample

The **Simple3DGameDX** sample game implements a simple first-person 3D shooting gallery, where the player fires balls at moving targets. Hitting each target awards a set number of points, and the player can progress through 6 levels of increasing challenge. At the end of the levels, the points are tallied, and the player is awarded a final score.

The sample demonstrates these game concepts.

- Interoperation between DirectX 11.1 and the Windows Runtime
- A first-person 3D perspective and camera
- Stereoscopic 3D effects
- Collision-detection between objects in 3D
- Handling player input for mouse, touch, and Xbox controller controls
- Audio mixing and playback
- A basic game state-machine



Topic	Description
Set up the game project	The first step in developing your game is to set up a project in Microsoft Visual Studio. After you've configured a project specifically for game development, you could later re-use it as a kind of template.
Define the game's UWP app framework	The first step in coding a Universal Windows Platform (UWP) game is building the framework that lets the app object interact with Windows.
Game flow management	Define the high-level state machine to enable player and system interaction. Learn how UI interacts with the overall game's state machine and how to create event handlers for UWP games.

Topic	Description
Define the main game object	Now, we look at the details of the sample game's main object and how the rules it implements translate into interactions with the game world.
Rendering framework I: Intro to rendering	Learn how to develop the rendering pipeline to display graphics. Intro to rendering.
Rendering framework II: Game rendering	Learn how to assemble the rendering pipeline to display graphics. Game rendering, set up and prepare data.
Add a user interface	Learn how to add a 2D user interface overlay to a DirectX UWP game.
Add controls	Now, we take a look at how the sample game implements move-look controls in a 3-D game, and how to develop basic touch, mouse, and game controller controls.
Add sound	Develop a simple sound engine using XAudio2 APIs to playback game music and sound effects.
Extend the sample game	Learn how to implement a XAML overlay for a UWP DirectX game.

Create a console calculator in C++

Article • 10/08/2024

The usual starting point for a C++ programmer is a "Hello, world!" application that runs on the command line. You start with that in this article, and then move on to something more challenging: a calculator app.

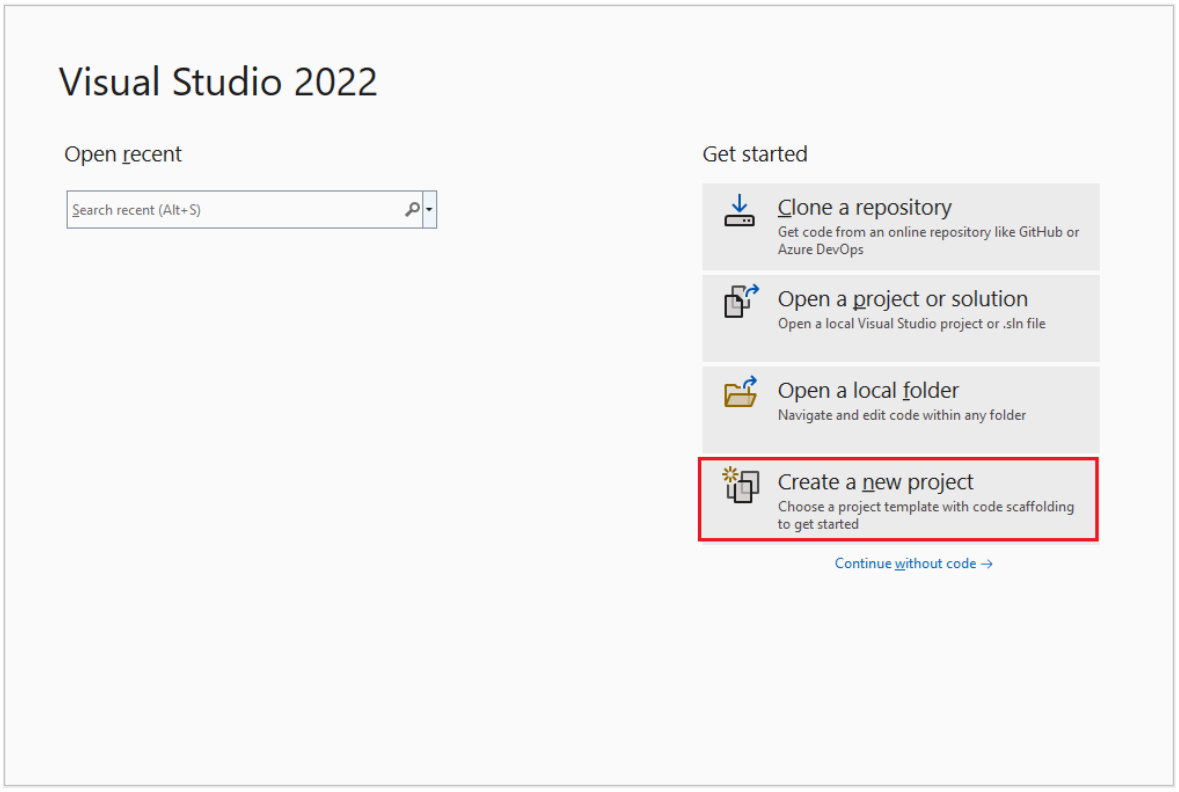
Prerequisites

- Visual Studio with the **Desktop development with C++** workload installed and running on your computer. To install it, see Install C++ support in Visual Studio.
- This tutorial demonstrates a feature called edit and continue which allows you to make changes to your code while the app is running. To enable edit and continue, from the main menu select **Tools** > **Options** > **Debugging** > **General** and ensure that **Require source files to exactly match the original version** is checked.

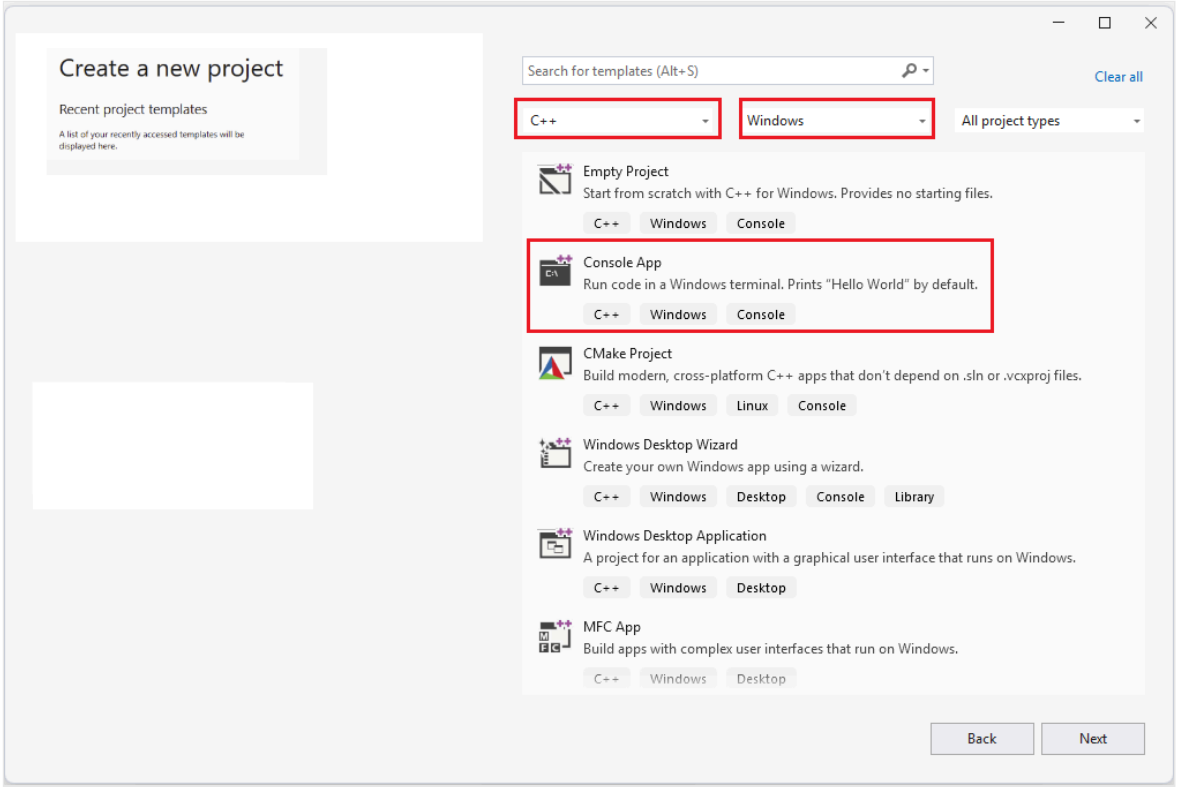
Create your app project

Visual Studio uses *projects* to organize the code for an app, and *solutions* to organize one or more projects. A project contains all the options, configurations, and rules used to build an app. It also manages the relationship between all the project's files and any external files. To create your app, first, create a new project and solution.

1. Start Visual Studio--the Visual Studio Start dialog box appears. Select **Create a new project** to get started.

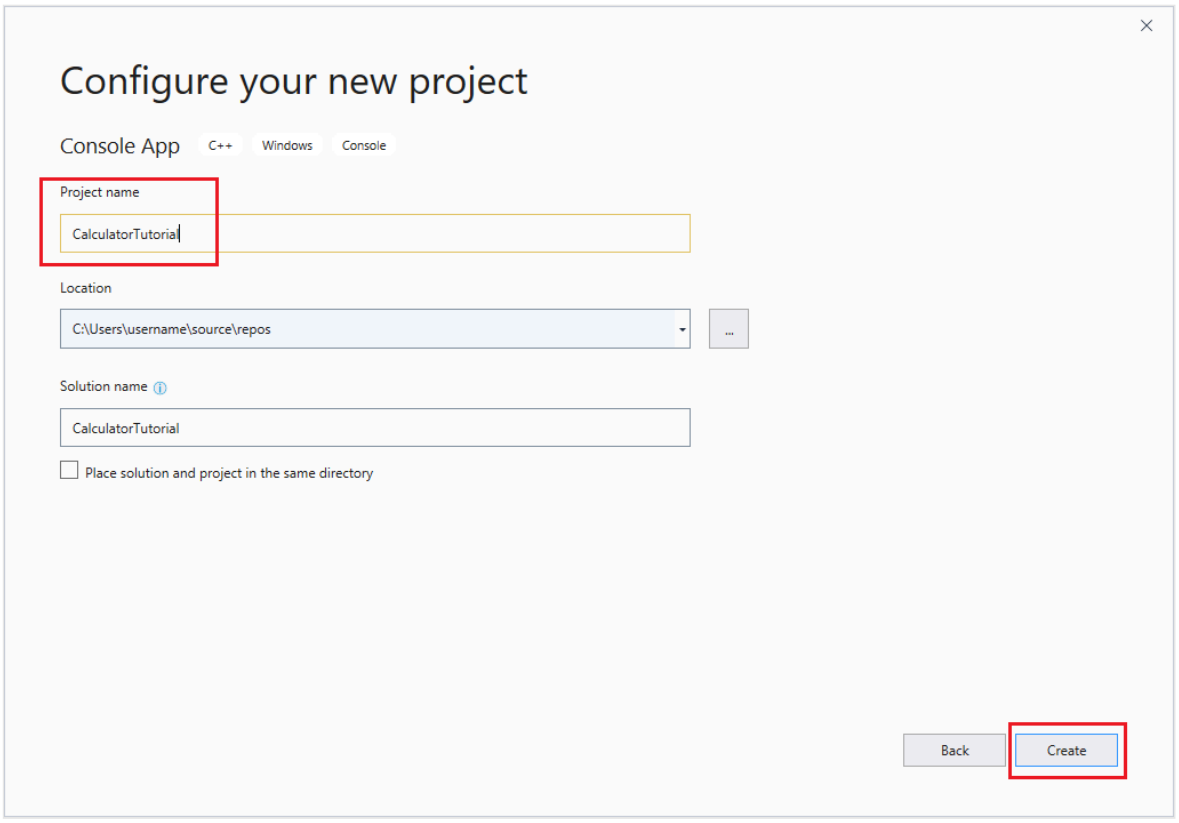


2. In the **Create a new project** dialog, set the language dropdown to **C++**, set the platform dropdown to **Windows**, select **Console App** from the list of project types, then select **Next**.



> **ⓘ Important**
>
> Make sure you select the C++ version of the **Console App** template. It has the **C++**, **Windows**, and **Console** tags, and the icon has "++" in the corner.

3. In the **Configure your new project** dialog box, select the **Project name** text box, name your new project *CalculatorTutorial*, then select **Create**.



An empty C++ Windows console application 'Hello World' app is created. Console applications use a Windows console window to display output and accept user input. In Visual Studio, an editor window opens and shows the generated code:

C++

```
// CalculatorTutorial.cpp : This file contains the 'main' function.
Program execution begins and ends there.
//

#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu

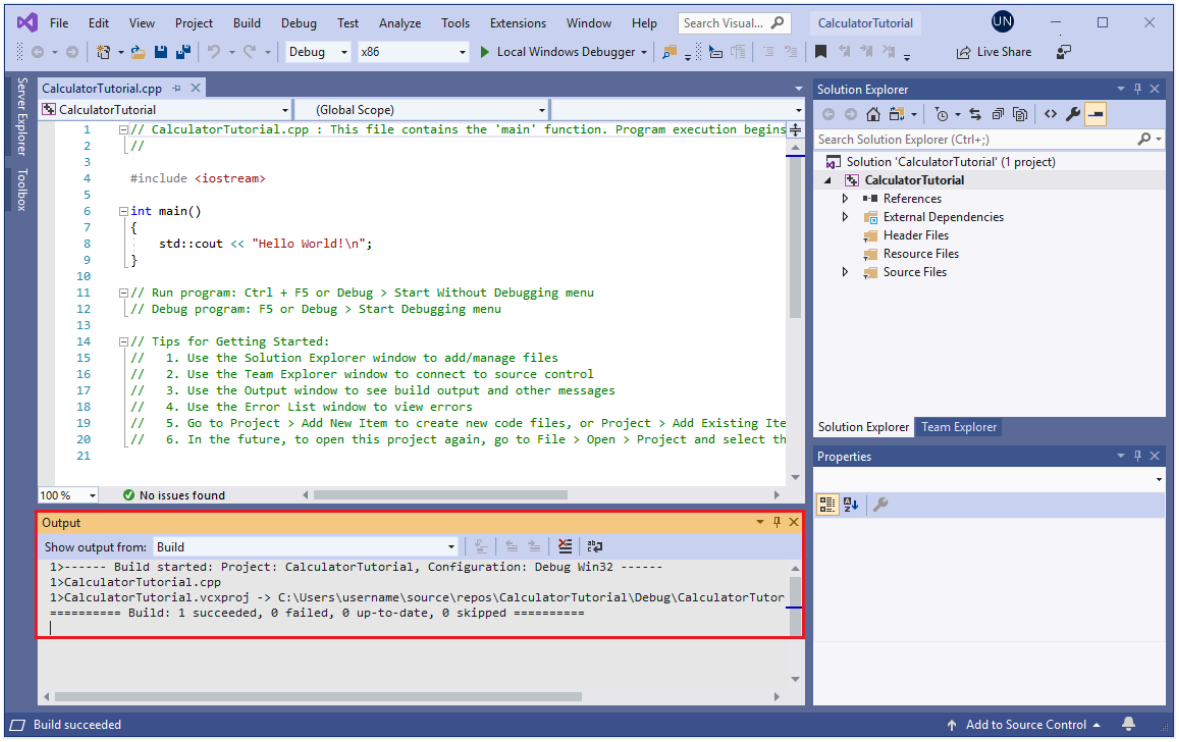
// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or
Project > Add Existing Item to add existing code files to the project
```

```
//   6. In the future, to open this project again, go to File > Open >
Project and select the .sln file
```

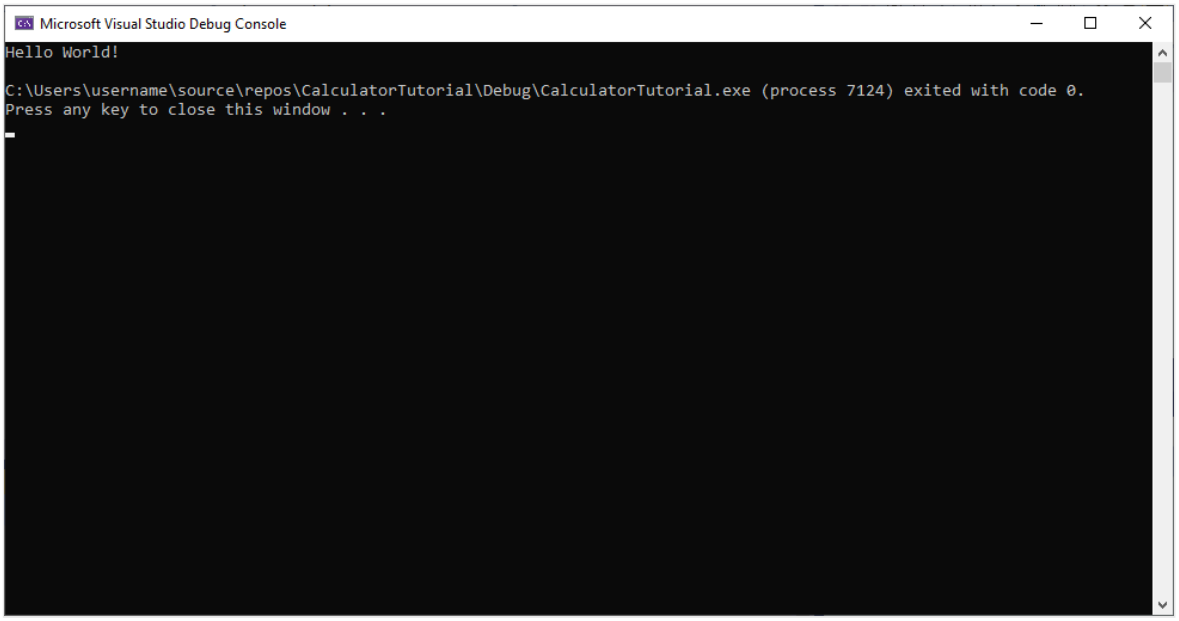
Verify that your new app builds and runs

The template for a new Windows console application creates a simple C++ "Hello World" app. At this point, you can see how Visual Studio builds and runs the apps you create right from the IDE.

1. To build your project, select **Build Solution** from the **Build** menu. The **Output** window shows the results of the build process.



2. To run the code, on the menu bar, select **Debug** > **Start without debugging** (Ctrl+F5).



A console window opens and your app runs within it.

When you start a console app in Visual Studio, it runs your code, then prints "Press any key to close this window . . ." to give you a chance to see the output.

Congratulations! You created your first "Hello, world!" console app in Visual Studio!

3. Press a key to dismiss the console window and return to Visual Studio.

You now have the tools to build and run your app after every change, to verify that the code still works as you expect. Later, we show you how to debug it if it doesn't.

Edit the code

Now let's modify the code in this template to be a calculator app.

1. Replace the contents of the *`CalculatorTutorial.cpp`* file with the following code so that it matches this example:

```
C++
```

```cpp
// CalculatorTutorial.cpp : This file contains the 'main' function.
Program execution begins and ends there.
//

#include <iostream>

using namespace std;

int main()
{
    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b |
```

```
a*b | a/b"
        << endl;
    return 0;
}

// Run program: Ctrl + F5 or Debug > Start Without Debugging menu
// Debug program: F5 or Debug > Start Debugging menu
// Tips for Getting Started:
//   1. Use the Solution Explorer window to add/manage files
//   2. Use the Team Explorer window to connect to source control
//   3. Use the Output window to see build output and other messages
//   4. Use the Error List window to view errors
//   5. Go to Project > Add New Item to create new code files, or
Project > Add Existing Item to add existing code files to the project
//   6. In the future, to open this project again, go to File > Open >
Project and select the .sln file
```

Understanding the code:

- The `#include` statement brings in code in other files. Sometimes, you may see a filename surrounded by angle brackets like `<iostream>`. The angle brackets instruct the compiler to look for the `iostream` header file first in the standard system directories, and if not found, to look in directories specific to the project. Other times, you may see a filename surrounded by quotes like `"someHeader.h"`. The quotes instruct the compiler to skip looking in the standard system directories and instead only look in directories specific to the project.
- The `using namespace std;` tells the compiler to expect code from the C++ Standard Library to be used in this file. Without this line, each keyword from the library would have to be preceded with `std::` to denote its scope. For instance, without that line, each reference to `cout` would be written as `std::cout`. The **`using`** statement is added to make it more convenient to access code in another namespace.
- The `cout` keyword is used to print to standard output in C++. The `<<` operator tells the compiler to send whatever is to the right of it to the standard output.
- The `endl` keyword is like the Enter key; it ends the line and moves the cursor to the next line. It's a better practice to put a `\n` inside the string (contained by `""`) to do the same thing because `endl` always flushes the buffer which can hurt the performance of the program. But since this is a very small app, `endl` is used instead.
- All C++ statements must end with semicolons and all C++ applications must contain a `main()` function. This function is what the program runs at

> the start. All code must be accessible from `main()` in order to be used.

2. To save the file, press **Ctrl+S**, or select the floppy disk icon in the toolbar under the menu bar.

3. To run the application, press **Ctrl+F5** or go to the **Debug** menu and select **Start Without Debugging**. You should see a console window appear that looks like this.

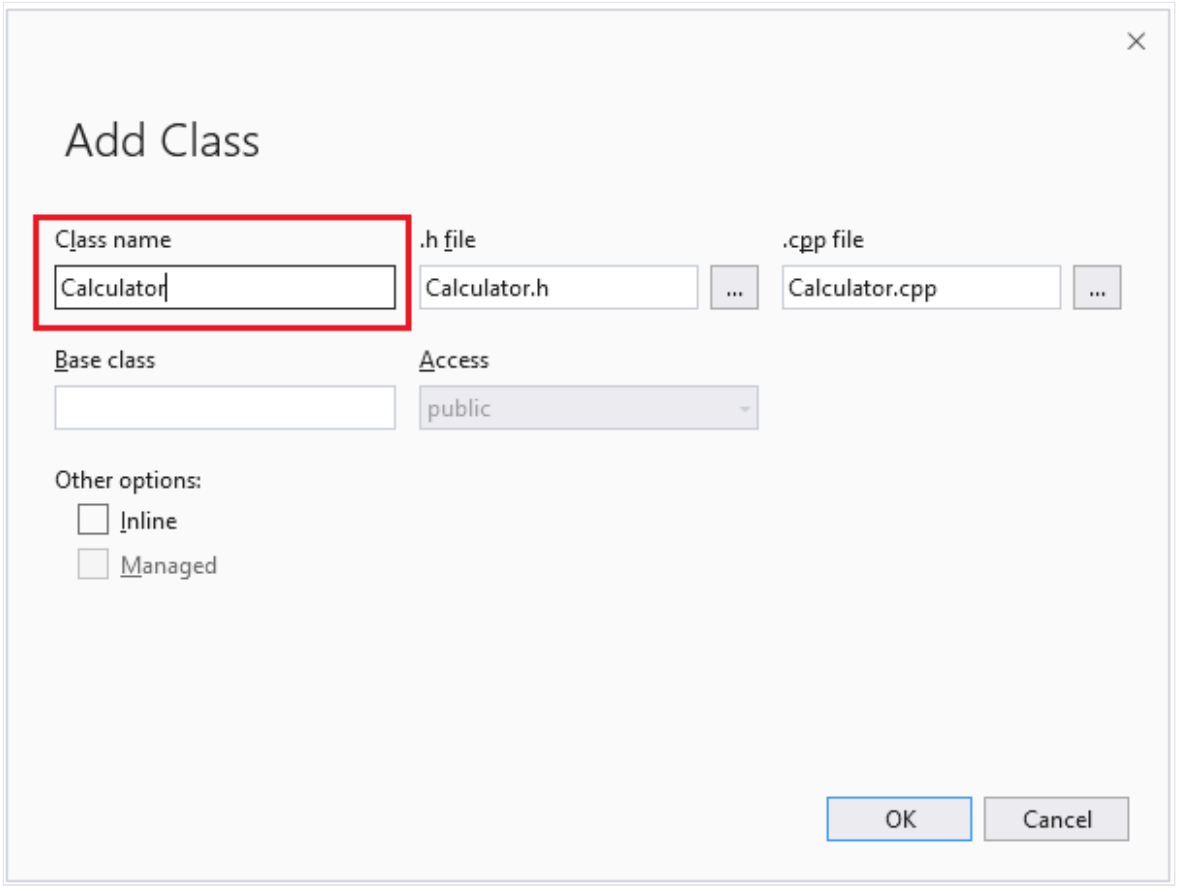
4. Close the console window when you're done.

Add code to do some math

A class is like a blueprint for an object that does something. In this case, we define a calculator class to contain the math logic.

Add a Calculator class

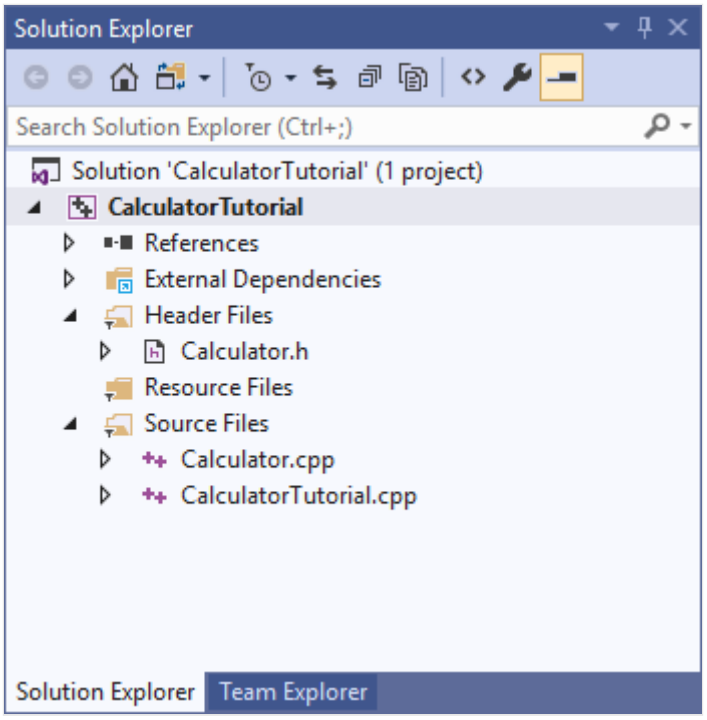
1. Go to the **Project** menu and select **Add Class**. In the **Class Name** edit box, enter *Calculator*. Select **OK**.



Two new files get added to your project. To save all your changed files at once, press **Ctrl+Shift+S**. It's a keyboard shortcut for **File** > **Save All**. There's also a toolbar button for **Save All**, an icon of two floppy disks, found beside the **Save**

button. In general, it's good practice to do **Save All** frequently, so you don't miss saving any changes.

The **Add Class** wizard creates `.h` and `.cpp` files that have the same name as the class. You can see a full list of your project files in the **Solution Explorer** window, visible on the side of the IDE. If the window isn't visible, open it from the menu bar via **View** > **Solution Explorer**.



You can open a file by double-clicking it in the **Solution Explorer** window. Double-click `Calculator.h` to open it.

2. Replace the contents of **`Calculator.h`** with the following code so that the file now looks like this:

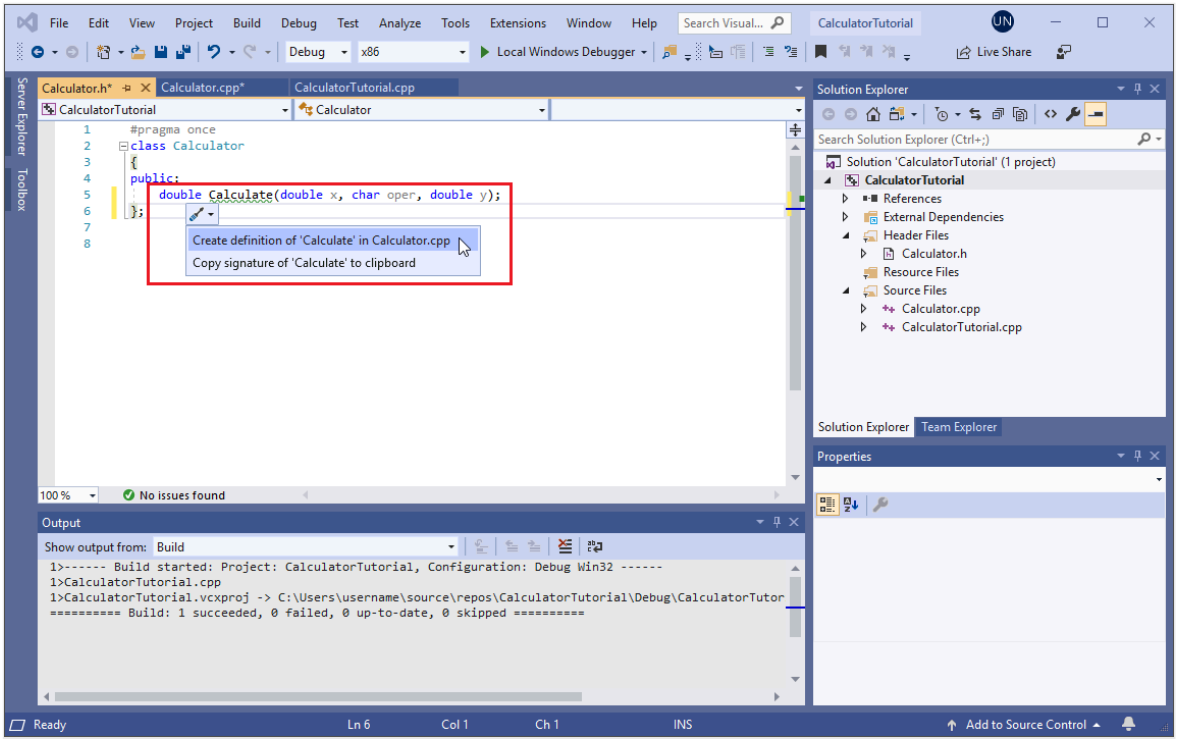
```
C++
```

```
#pragma once
class Calculator
{
public:
    double Calculate(double x, char oper, double y);
};
```

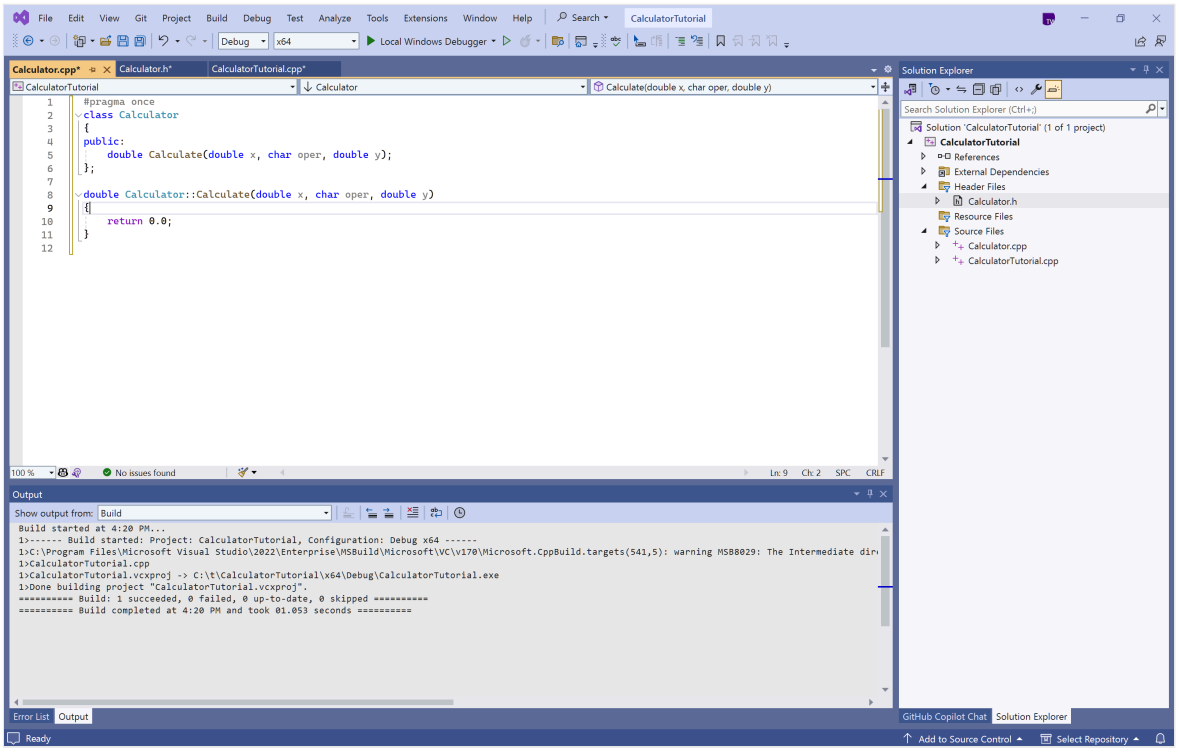
> Understanding the code
>
> - This code declares a new function called `Calculate`, which handles math operations for addition, subtraction, multiplication, and division.

- C++ code is organized into *header* (`.h`) files and *source* (`.cpp`) files. Some other file extensions are supported by various compilers, but these are the main ones to know about. Functions and variables are normally *declared*, that is, given a name and a type, in header files, and *implemented*, or given a definition, in source files. To access code defined in another file, you can use `#include "filename.h"`, where `filename.h` is the name of the file that declares the variables or functions you want to use.
- It's good practice to organize your code into different files based on what it does, so it's easy to find the code you need later. In our case, we define the `Calculator` class separately from the file containing the `main()` function, but we plan to reference the `Calculator` class in `main()`.

3. A green squiggle appears under `Calculate` because although the `Calculate` function is *declared*, it isn't *defined*. Hover over `Calculate`, click the down arrow on the screwdriver icon, and select **Create definition of 'Calculate' in `Calculator.cpp`**.



This code is added to `Calculator.cpp`:



Currently, it just returns 0.0. Let's change that.

4. Switch to the `Calculator.cpp` file in the editor window. Replace the contents of `Calculator::Calculate(double x, char oper, double y)` with:

C++

```
double Calculator::Calculate(double x, char oper, double y)
{
    switch(oper)
    {
        case '+':
            return x + y;
        case '-':
            return x - y;
        case '*':
            return x * y;
        case '/':
            return x / y;
        default:
            return 0.0;
    }
}
```

> Understanding the code
>
> - The function `Calculate` takes a number, an operator, and a second number. Then it performs the requested operation on the two numbers.
> - The `switch` statement checks which operator was provided, and executes the case corresponding to that operation. The `default:` case is a fallback

in case the user types an operator that isn't handled by any of the preceding `case` statements. It's best to handle invalid user input in a more elegant way, but this is beyond the scope of this tutorial.

- The **`double`** keyword denotes a type of number that supports decimals. This type of number is called a floating-point number, and `double` means a floating point number that has extra precision. This way, the calculator can handle both decimal math and integer math. The `Calculate` function is required to always return a double-precision floating point number due to the **`double`** at the start of the code (this denotes the function's return type), which is why we return 0.0 in the default case.
- The `.h` file declares the function *prototype*, which tells the compiler upfront what parameters it requires, and what return type to expect from it. The `.cpp` file has all the implementation details of the function.

If you build and run the code again at this point, it immediately exits after asking which operation to perform. So, modify the `main` function to do multiple calculations.

Call the `Calculator` class member functions

1. Update the `main` function in *`CalculatorTutorial.cpp`* as follows:

C++

```
// CalculatorTutorial.cpp : This file contains the 'main' function.
Program execution begins and ends there.
//

#include <iostream>
#include "Calculator.h"

using namespace std;

int main()
{
    double x = 0.0;
    double y = 0.0;
    double result = 0.0;
    char oper = '+';

    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b |
a*b | a/b"
         << endl;

    Calculator c;
    while (true)
    {
```

```
        cin >> x >> oper >> y;
        result = c.Calculate(x, oper, y);
        cout << "Result " << "of " << x << oper << y << " is: " <<
result << endl;
    }

    return 0;
}
```

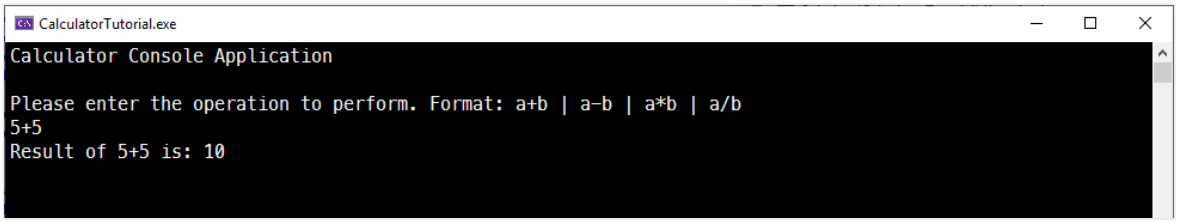
> Understanding the code
>
> - Since C++ programs always start at the `main()` function, we need to call our other code from there, so an `#include` statement is needed to make that code visible to our `main()` function.
> - The variables `x`, `y`, `oper`, and `result` are declared to store the first number, second number, operator, and final result, respectively. It's always good practice to give them some initial values to avoid undefined behavior, which is what is done here.
> - The `Calculator c;` line declares an object named `c` as an instance of the `Calculator` class. The class itself is just a blueprint for how calculators work; the object is the specific calculator that does the math.
> - The `while (true)` statement is a loop. The code inside the loop executes over and over again as long as the condition inside the `()` holds true. Since the condition is simply listed as **`true`**, it's always true, so the loop runs forever. To close the program, the user must manually close the console window. Otherwise, the program always waits for new input.
> - The `cin` keyword accepts input from the user. The input stream is smart enough to process a line of text entered in the console window and place it inside each of the variables listed, in order.
> - The `c.Calculate(x, oper, y);` expression calls the `Calculate` function defined earlier, and supplies the entered input values and the requested operation. The function then returns a number that is stored in `result`.
> - Finally, `result` is printed to the console and the user sees the result of the calculation.

Build and test the code again

Now test the program again to make sure everything works properly.

1. Press **Ctrl+F5** to rebuild and start the app.

2. Enter `5+5`, and press **Enter**. Verify that the result is 10.



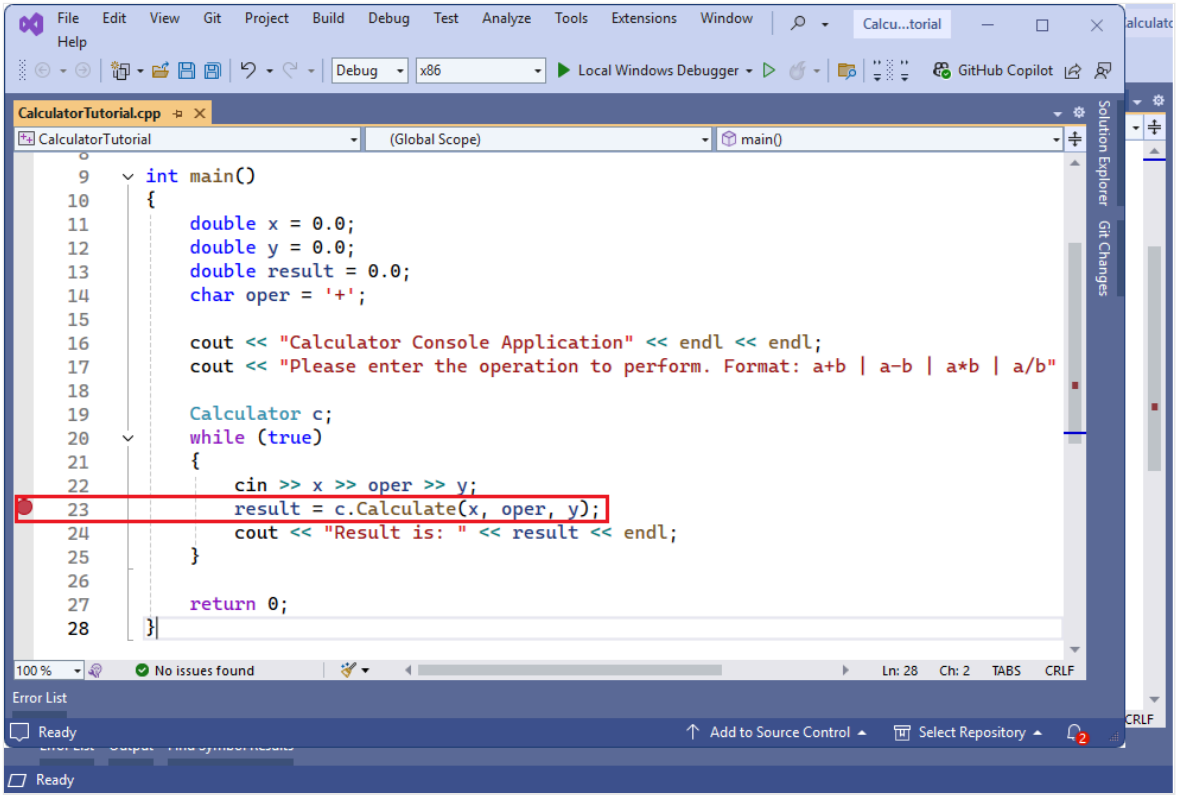
3. Stop the program by closing the console window.

Debug the app

Since the user is free to type anything into the console window, let's make sure the calculator handles unexpected input. Instead of running the program, let's debug it so we can inspect what it's doing step-by-step.

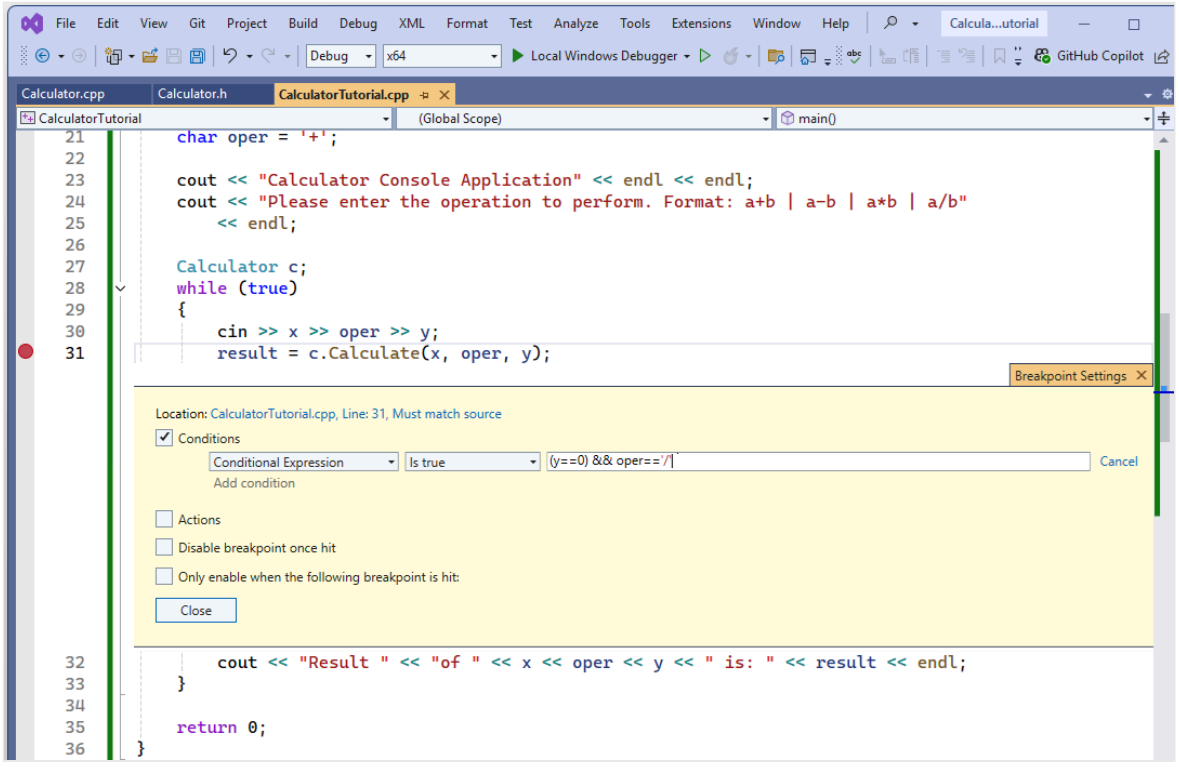
Run the app in the debugger

1. In `CalcuatorTutorial.cpp`, set a breakpoint on the line: `result = c.Calculate(x, oper, y);`. To set the breakpoint, click next to the line in the gray vertical bar along the left edge of the editor window so that a red dot appears.



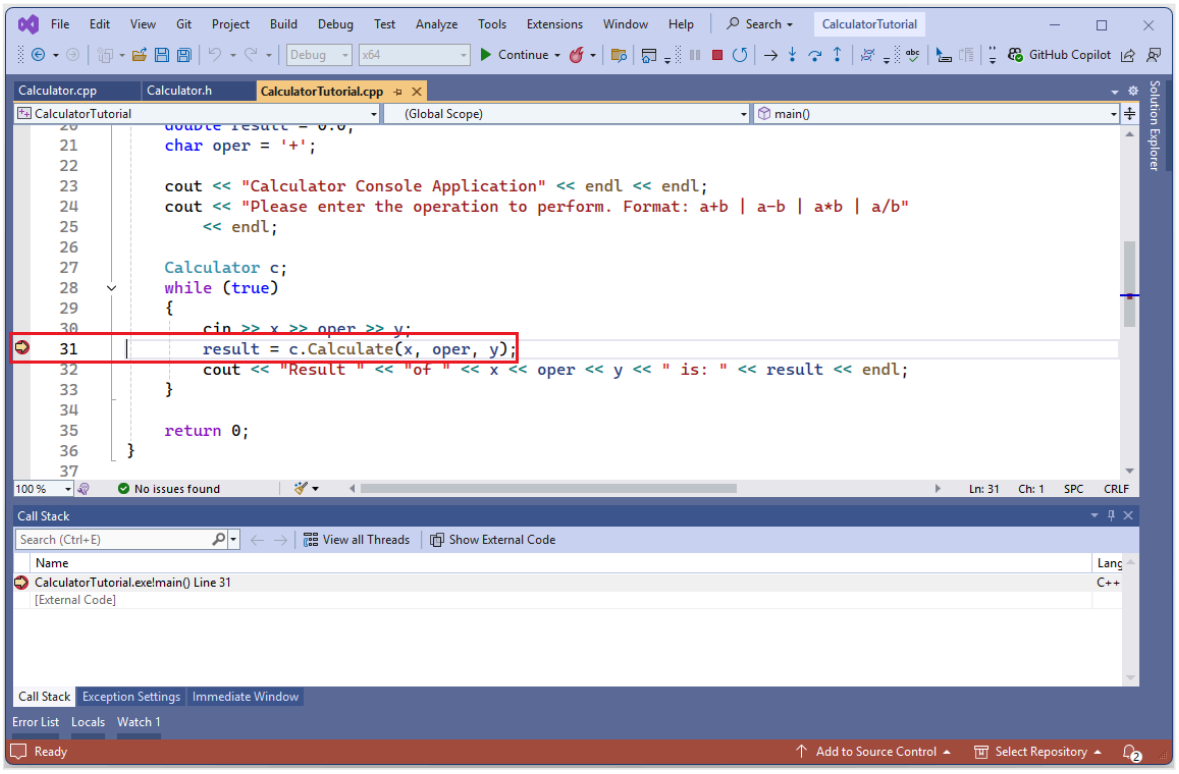
Now when we debug the program, execution pauses at that line. We already have a rough idea that the program works for simple cases. Since we don't want to pause execution every time we call `Calculate()`, let's make the breakpoint conditional.

2. Right-click the red dot that represents the breakpoint, and select **Conditions**. In the edit box for the condition, enter `(y == 0) && (oper == '/')`. Select the **Close** button to save the breakpoint condition.



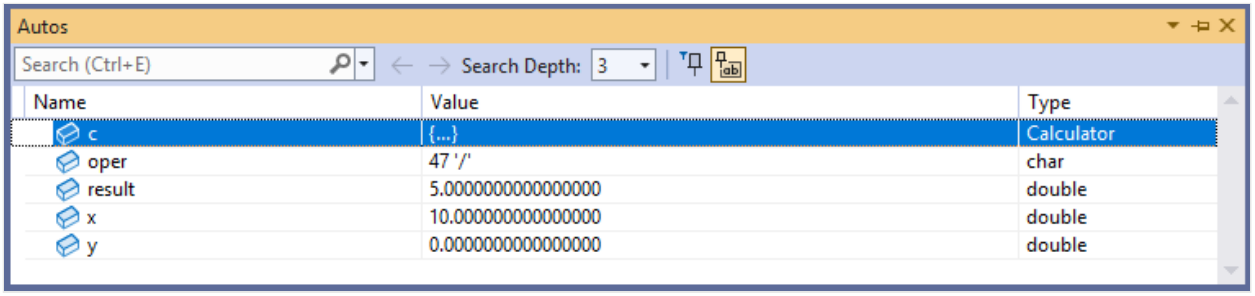
Now, execution pauses at the breakpoint when the app tries to divide by 0.

3. To debug the program, press **F5**, or select the **Local Windows Debugger** debugger toolbar button that has the green arrow icon. In your console app, if you enter something like "5 - 0", the program behaves normally and keeps running. However, if you type "10 / 0", it pauses at the breakpoint. You can put any number of spaces between the operator and numbers: `cin` is smart enough to parse the input appropriately.

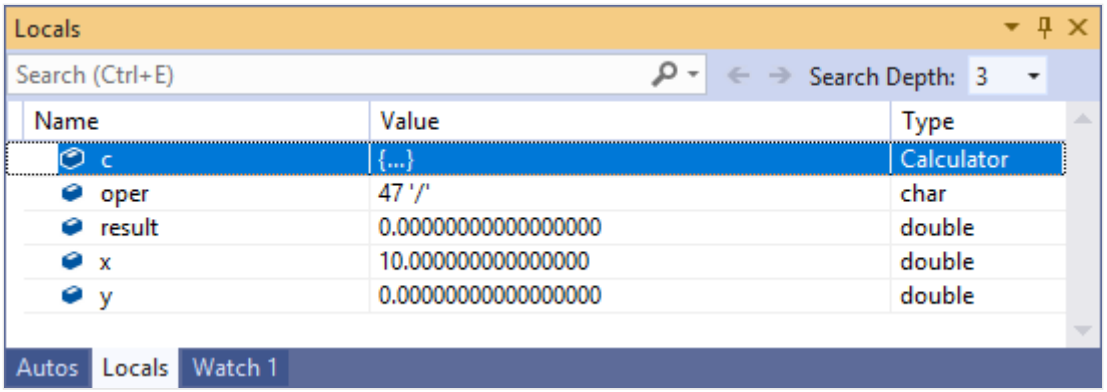


Useful windows in the debugger

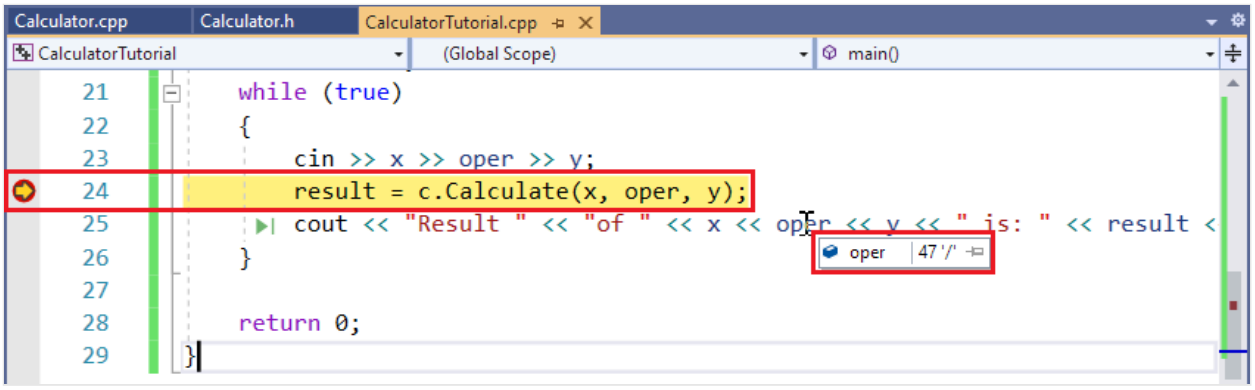
When you debug your code, you may notice that some new windows appear. These windows can assist your debugging experience. Take a look at the **Autos** window. The **Autos** window shows you the current values of variables used at least three lines before and up to the current line. If you don't see the **Autos** window, from the main menu select **Debug** > **Windows** > **Autos**.



To see all of the variables from that function, switch to the **Locals** window. Because this is a small function, the Autos and Locals window show the same variables. But you can modify the values of these variables in the Locals window while debugging to see what effect they would have on the program. In this case, we leave them alone. Open the **Locals** window by selecting **Locals** at the bottom of the **Autos** window, or by selecting from the main menu **Debug** > **Windows** > **Locals**.



You can also hover over variables in the code to see their current values at the point where execution is currently paused. Make sure the editor window is in focus by clicking on it first.



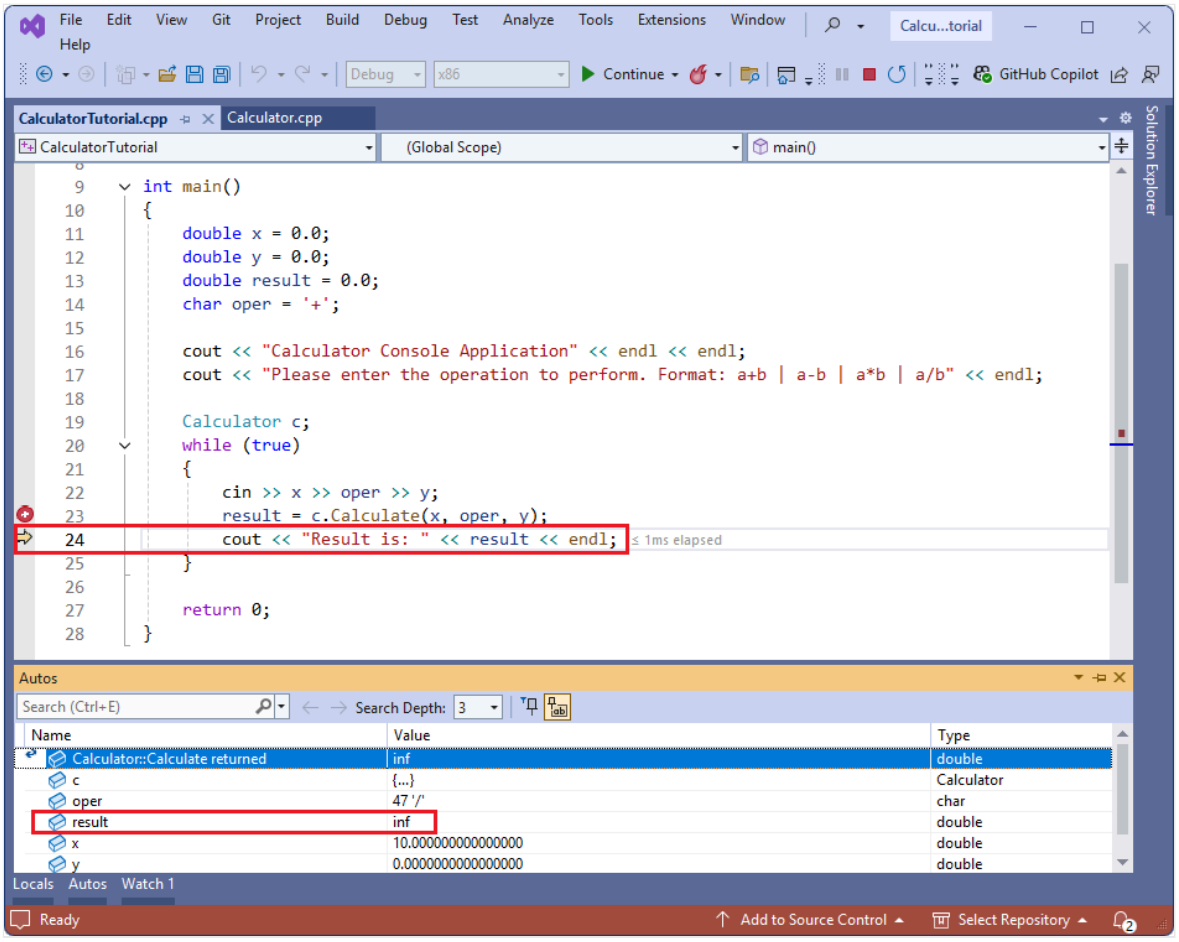
Continue debugging

1. The yellow arrow on the left shows the current point of execution. The current line calls `Calculate`, so press **F11** to **Step Into** the function. Now you're executing code in the body of the `Calculate` function. Be careful with **Step Into** because it steps into any functions on the line you're on, including standard library functions. It's fine to step into the standard library, but you may be more interested in focusing on your code instead of library code.

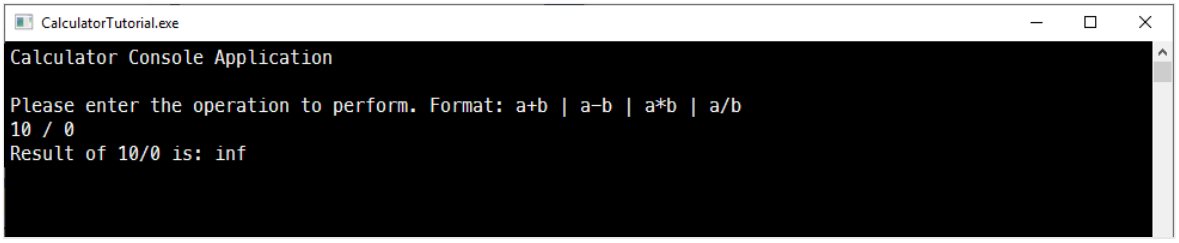
2. Now that the point of execution is at the start of the `Calculate` function, press **F10** to move to the next line in the program's execution. **F10** is also known as **Step Over**. You can use **Step Over** to move from line to line, without delving into the details of what is occurring in each part of the line. In general, you should use **Step Over** instead of **Step Into** unless you want to dive more deeply into code that is being called from elsewhere (as you did to reach the body of `Calculate`).

3. Continue using **F10** to **Step Over** each line until you get back to the `main()` function in the other file, and stop on the `cout` line.

The program is doing what's expected: it takes the first number, and divides it by the second. On the `cout` line, hover over the `result` variable or take a look at `result` in the **Autos** window. Its value is `inf`, which doesn't look right.



Let's fix it. The `cout` line outputs whatever value is stored in `result`, so when you step one more line forward using **F10**, the console window displays:



This result is because division by zero is undefined, so the program doesn't have a numerical answer for the requested operation.

Fix the "divide by zero" error

Let's handle division by zero more gracefully so that it's easier for the user to understand the problem.

1. Make the following changes in *`CalculatorTutorial.cpp`*. You can leave the program running as you edit, thanks to a debugger feature called **Edit and**

Continue. Add an `if` statement following `cin >> x >> oper >> y;` to check for division by zero and output a message to the user if it happens. Otherwise, the result is printed.

C++

```
// CalculatorTutorial.cpp : This file contains the 'main' function.
Program execution begins and ends there.
//

#include <iostream>
#include "Calculator.h"

using namespace std;

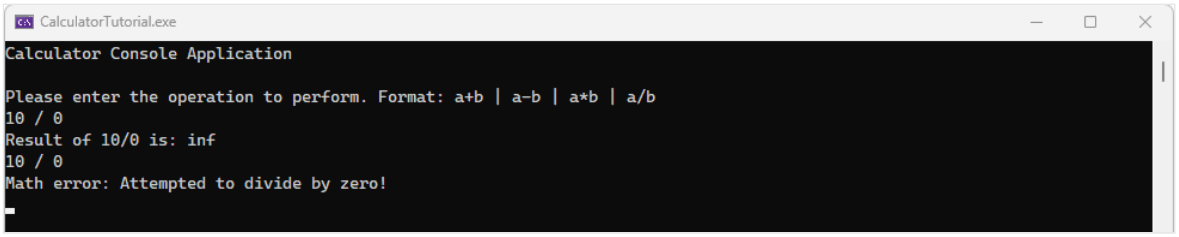
int main()
{
    double x = 0.0;
    double y = 0.0;
    double result = 0.0;
    char oper = '+';

    cout << "Calculator Console Application" << endl << endl;
    cout << "Please enter the operation to perform. Format: a+b | a-b |
a*b | a/b" << endl;

    Calculator c;
    while (true)
    {
        cin >> x >> oper >> y;
        if (oper == '/' && y == 0)
        {
            cout << "Math error: Attempted to divide by zero!" << endl;
            continue;
        }
        else
        {
            result = c.Calculate(x, oper, y);
        }
        cout << "Result " << "of " << x << oper << y << " is: " <<
result << endl;
    }

    return 0;
}
```

2. Press **F5** once. Program execution continues until it has to pause to ask for user input. Enter `10 / 0` again. Now, a more helpful message is printed. The user is asked for more input, and the program continues executing normally.



> **Note**
>
> When you edit code while in debugging mode, there's a risk of code becoming stale. This happens when the debugger is still running your old code, and has not yet updated it with your changes. The debugger displays a dialog to inform you when this happens. Sometimes, you may need to press **F5** to refresh the code being executed. In particular, if you make a change inside a function while the point of execution is inside that function, you need to step out of the function, then back into it again to get the updated code. If that doesn't work and you see an error message, you can stop debugging by clicking on the red square in the toolbar under the menus at the top of the IDE, then start debugging again by entering **F5** or by choosing the green "play" arrow beside the stop button on the toolbar.
>
> Another reason edit and continue may fail is if you see a message that says "The Require source files to exactly match the original version setting under Debug->Options->General needs to be enabled..." To fix this, from the main menu select **Tools** > **Options** > **Debugging** > **General** and ensure that **Require source files to exactly match the original version** is checked.

> Understanding the Run and Debug shortcuts
>
> - **F5**, or **Debug** > **Start Debugging**, starts a debugging session, if one isn't already active, and runs the program until a breakpoint is hit or the program needs user input. If no user input is needed and no breakpoint is available to hit, the program terminates and the console window closes itself when the program finishes running. If your program outputs to the console, use **Ctrl+F5** or set a breakpoint before you press **F5** to keep the window open.
> - **Ctrl+F5**, or **Debug** > **Start Without Debugging**, runs the application without going into debug mode. This is slightly faster than debugging, and the console window stays open after the program finishes executing.
> - **F10**, known as **Step Over**, lets you iterate through code, line-by-line, and visualize how the code is run and what variable values are at each step of execution.

> - **F11**, known as **Step Into**, works similarly to **Step Over**, except it steps into any functions called on the line of execution. For example, if the line being executed calls a function, pressing **F11** moves the pointer into the body of the function, so you can follow the function's code being run before coming back to the line you started at. Pressing **F10** steps over the function call and just moves to the next line; the function call still happens, but the program doesn't pause to show you what it's doing.

Close the app

- If it's still running, close the console window to stop the calculator app.

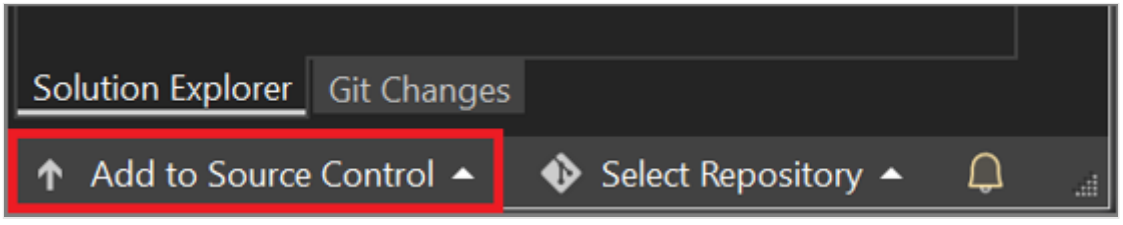
Add Git source control

Now that you've created an app, you might want to add it to a Git repository. We've got you covered. Visual Studio makes that process easy with Git tools you can use directly from the IDE.

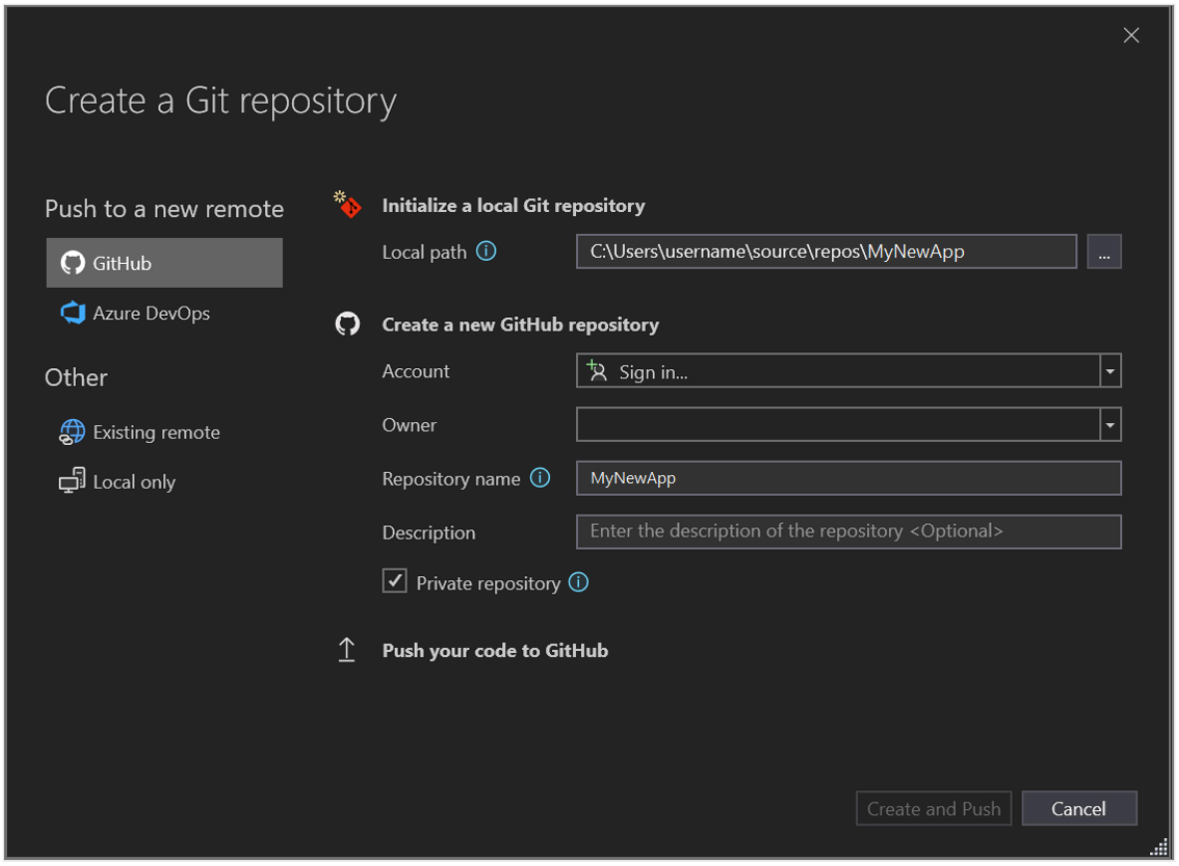
> **Tip**
>
> Git is the most widely used modern version control system, so whether you're a professional developer or you're learning how to code, Git can be very useful. If you're new to Git, the https://git-scm.com/ website is a good place to start. There, you can find cheat sheets, a popular online book, and Git Basics videos.

To associate your code with Git, you start by creating a new Git repository where your code is located. Here's how:

1. In the status bar at the bottom-right corner of Visual Studio, select **Add to Source Control**, and then select **Git**.



2. In the **Create a Git repository** dialog box, sign in to GitHub.

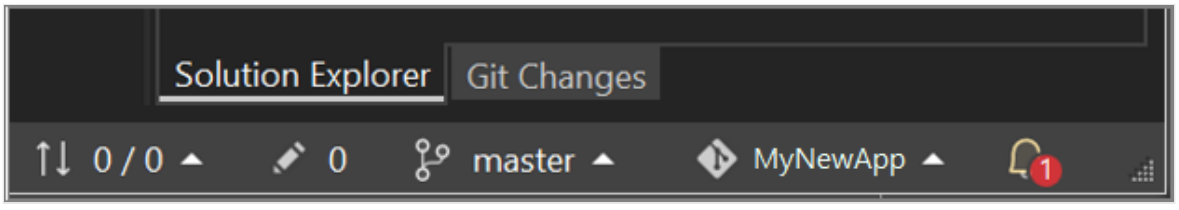


The repository name auto-populates based on your folder location. By default, your new repository is private, which means you're the only one who can access it.

> **Tip**
>
> Whether your repository is public or private, it's best to have a remote backup of your code stored securely on GitHub. Even if you aren't working with a team, a remote repository makes your code available to you from any computer.

3. Select **Create and Push**.

After you create your repository, status details appear in the status bar.



The first icon with the arrows shows how many outgoing/incoming commits are in your current branch. You can use this icon to pull any incoming commits or push any outgoing commits. You can also choose to view these commits first. To do so, select the icon, and then select **View Outgoing/Incoming**.

The second icon with the pencil shows the number of uncommitted changes to your code. You can select this icon to view those changes in the **Git Changes** window.

To learn more about how to use Git with your app, see the Visual Studio version control documentation.

The finished app

Congratulations! You completed the code for the calculator app, built and debugged it, and added it to a repo, all in Visual Studio.

Next steps

Learn more about Visual Studio for C++

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Get started with C++/WinRT

Article • 02/13/2023

> **Important**
>
> For info about setting up Visual Studio for C++/WinRT development—including installing and using the C++/WinRT Visual Studio Extension (VSIX) and the NuGet package (which together provide project template and build support)—see **Visual Studio support for C++/WinRT**.

To get you up to speed with using C++/WinRT, this topic walks through a simple code example based on a new **Windows Console Application (C++/WinRT)** project. This topic also shows how to add C++/WinRT support to a Windows Desktop application project.

> **Note**
>
> While we recommend that you develop with the latest versions of Visual Studio and the Windows SDK, if you're using Visual Studio 2017 (version 15.8.0 or later), and targeting the Windows SDK version 10.0.17134.0 (Windows 10, version 1803), then a newly created C++/WinRT project may fail to compile with the error "*error C3861: 'from_abi': identifier not found*", and with other errors originating in *base.h*. The solution is to either target a later (more conformant) version of the Windows SDK, or set project property **C/C++** > **Language** > **Conformance mode: No** (also, if **/permissive-** appears in project property **C/C++** > **Language** > **Command Line** under **Additional Options**, then delete it).

A C++/WinRT quick-start

Create a new **Windows Console Application (C++/WinRT)** project.

Edit `pch.h` and `main.cpp` to look like this.

C++/WinRT

```
// pch.h
#pragma once
#include <winrt/Windows.Foundation.Collections.h>
```

```
#include <winrt/Windows.Web.Syndication.h>
#include <iostream>
```

C++/WinRT

```
// main.cpp
#include "pch.h"

using namespace winrt;
using namespace Windows::Foundation;
using namespace Windows::Web::Syndication;

int main()
{
    winrt::init_apartment();

    Uri rssFeedUri{ L"https://blogs.windows.com/feed" };
    SyndicationClient syndicationClient;
    syndicationClient.SetRequestHeader(L"User-Agent", L"Mozilla/5.0
(compatible; MSIE 10.0; Windows NT 6.2; WOW64; Trident/6.0)");
    SyndicationFeed syndicationFeed =
syndicationClient.RetrieveFeedAsync(rssFeedUri).get();
    for (const SyndicationItem syndicationItem : syndicationFeed.Items())
    {
        winrt::hstring titleAsHstring = syndicationItem.Title().Text();

        // A workaround to remove the trademark symbol from the title
string, because it causes issues in this case.
        std::wstring titleAsStdWstring{ titleAsHstring.c_str() };
        titleAsStdWstring.erase(remove(titleAsStdWstring.begin(),
titleAsStdWstring.end(), L'™'), titleAsStdWstring.end());
        titleAsHstring = titleAsStdWstring;

        std::wcout << titleAsHstring.c_str() << std::endl;
    }
}
```

Let's take the short code example above piece by piece, and explain what's going on in each part.

C++/WinRT

```
#include <winrt/Windows.Foundation.Collections.h>
#include <winrt/Windows.Web.Syndication.h>
```

With the default project settings, the included headers come from the Windows SDK, inside the folder `%WindowsSdkDir%Include<WindowsTargetPlatformVersion>\cppwinrt\winrt`. Visual Studio includes that path in its *IncludePath* macro. But there's no strict dependency on the

Windows SDK, because your project (via the `cppwinrt.exe` tool) generates those same headers into your project's *$(GeneratedFilesDir)* folder. They'll be loaded from that folder if they can't be found elsewhere, or if you change your project settings.

The headers contain Windows APIs projected into C++/WinRT. In other words, for each Windows type, C++/WinRT defines a C++-friendly equivalent (called the *projected type*). A projected type has the same fully-qualified name as the Windows type, but it's placed in the C++ **winrt** namespace. Putting these includes in your precompiled header reduces incremental build times.

> **ⓘ Important**
>
> Whenever you want to use a type from a Windows namespaces, you must `#include` the corresponding C++/WinRT Windows namespace header file, as shown above. The *corresponding* header is the one with the same name as the type's namespace. For example, to use the C++/WinRT projection for the Windows::Foundation::Collections::PropertySet runtime class, include the `winrt/Windows.Foundation.Collections.h` header.
>
> It is common for a C++/WinRT projection header to automatically include related namespace header files. For example, `winrt/Windows.Foundation.Collections.h` includes `winrt/Windows.Foundation.h`. But you shouldn't rely on this behavior, since it's an implementation detail that changes over time. You must explicitly include any headers that you need.

C++/WinRT

```
using namespace winrt;
using namespace Windows::Foundation;
using namespace Windows::Web::Syndication;
```

The `using namespace` directives are optional, but convenient. The pattern shown above for such directives (allowing unqualified name lookup for anything in the **winrt** namespace) is suitable for when you're beginning a new project and C++/WinRT is the only language projection you're using inside of that project. If, on the other hand, you're mixing C++/WinRT code with C++/CX and/or SDK application binary interface (ABI) code (you're either porting from, or interoperating with, one or both of those models), then see the topics Interop between C++/WinRT and C++/CX, Move to C++/WinRT from C++/CX, and Interop between C++/WinRT and the ABI.

C++/WinRT

```
winrt::init_apartment();
```

The call to **winrt::init_apartment** initializes the thread in the Windows Runtime; by default, in a multithreaded apartment. The call also initializes COM.

C++/WinRT

```
Uri rssFeedUri{ L"https://blogs.windows.com/feed" };
SyndicationClient syndicationClient;
```

Stack-allocate two objects: they represent the uri of the Windows blog, and a syndication client. We construct the uri with a simple wide string literal (see String handling in C++/WinRT for more ways you can work with strings).

C++/WinRT

```
SyndicationFeed syndicationFeed =
syndicationClient.RetrieveFeedAsync(rssFeedUri).get();
```

SyndicationClient::RetrieveFeedAsync is an example of an asynchronous Windows Runtime function. The code example receives an asynchronous operation object from **RetrieveFeedAsync**, and it calls **get** on that object to block the calling thread and wait for the result (which is a syndication feed, in this case). For more about concurrency, and for non-blocking techniques, see Concurrency and asynchronous operations with C++/WinRT.

C++/WinRT

```
for (const SyndicationItem syndicationItem : syndicationFeed.Items()) { ...
}
```

SyndicationFeed.Items is a range, defined by the iterators returned from **begin** and **end** functions (or their constant, reverse, and constant-reverse variants). Because of this, you can enumerate **Items** with either a range-based `for` statement, or with the **std::for_each** template function. Whenever you iterate over a Windows Runtime collection like this, you'll need to `#include <winrt/Windows.Foundation.Collections.h>`.

C++/WinRT

```
winrt::hstring titleAsHstring = syndicationItem.Title().Text();

// Omitted: there's a little bit of extra work here to remove the trademark
symbol from the title text.
```

```
std::wcout << titleAsHstring.c_str() << std::endl;
```

Gets the feed's title text, as a **winrt::hstring** object (more details in String handling in C++/WinRT). The **hstring** is then output, via the **c_str** function, which reflects the pattern used with C++ Standard Library strings.

As you can see, C++/WinRT encourages modern, and class-like, C++ expressions such as `syndicationItem.Title().Text()`. This is a different, and cleaner, programming style from traditional COM programming. You don't need to directly initialize COM, nor work with COM pointers.

Nor do you need to handle HRESULT return codes. C++/WinRT converts error HRESULTs to exceptions such as **winrt::hresult-error** for a natural and modern programming style. For more info about error-handling, and code examples, see Error handling with C++/WinRT.

Modify a Windows Desktop application project to add C++/WinRT support

Some desktop projects (for example, the WinUI 3 templates in Visual Studio) have C++/WinRT support built in.

But this section shows you how you can add C++/WinRT support to any Windows Desktop application project that you might have. If you don't have an existing Windows Desktop application project, then you can follow along with these steps by first creating one. For example, open Visual Studio and create a **Visual C++** > **Windows Desktop** > **Windows Desktop Application** project.

You can optionally install the C++/WinRT Visual Studio Extension (VSIX) and the NuGet package. For details, see Visual Studio support for C++/WinRT.

Set project properties

Go to project property **General** > **Windows SDK Version**, and select **All Configurations** and **All Platforms**. Ensure that **Windows SDK Version** is set to 10.0.17134.0 (Windows 10, version 1803) or greater.

Confirm that you're not affected by Why won't my new project compile?.

Because C++/WinRT uses features from the C++17 standard, set project property **C/C++** > **Language** > **C++ Language Standard** to *ISO C++17 Standard (/std:c++17)*.

The precompiled header

The default project template creates a precompiled header for you, named either `framework.h`, or `stdafx.h`. Rename that to `pch.h`. If you have a `stdafx.cpp` file, then rename that to `pch.cpp`. Set project property **C/C++** > **Precompiled Headers** > **Precompiled Header** to *Create (/Yc)*, and **Precompiled Header File** to *pch.h*.

Find and replace all `#include "framework.h"` (or `#include "stdafx.h"`) with `#include "pch.h"`.

In `pch.h`, include `winrt/base.h`.

C++/WinRT

```
// pch.h
...
#include <winrt/base.h>
```

Linking

The C++/WinRT language projection depends on certain Windows Runtime free (non-member) functions, and entry points, that require linking to the WindowsApp.lib umbrella library. This section describes three ways of satisfying the linker.

The first option is to add to your Visual Studio project all of the C++/WinRT MSBuild properties and targets. To do this, install the Microsoft.Windows.CppWinRT NuGet package into your project. Open the project in Visual Studio, click **Project** > **Manage NuGet Packages...** > **Browse**, type or paste **Microsoft.Windows.CppWinRT** in the search box, select the item in search results, and then click **Install** to install the package for that project.

You can also use project link settings to explicitly link `WindowsApp.lib`. Or, you can do it in source code (in `pch.h`, for example) like this.

C++/WinRT

```
#pragma comment(lib, "windowsapp")
```

You can now compile and link, and add C++/WinRT code to your project (for example, code similar to that shown in the A C++/WinRT quick-start section, above).

The three main scenarios for C++/WinRT

As you use and become familiar with C++/WinRT, and work through the rest of the documentation here, you'll likely notice that there are three main scenarios, as described in the following sections.

Consuming Windows APIs and types

In other words, *using*, or *calling* APIs. For example, making API calls to communicate using Bluetooth; to stream and present video; to integrate with the Windows shell; and so on. C++/WinRT fully and uncompromisingly supports this category of scenario. For more info, see Consume APIs with C++/WinRT.

Authoring Windows APIs and types

In other words, *producing* APIs and types. For example, producing the kinds of APIs described in the section above; or the graphics APIs; the storage and file system APIs; the networking APIs, and so on. For more info, see Author APIs with C++/WinRT.

Authoring APIs with C++/WinRT is a little more involved than consuming them, because you must use IDL to define the shape of the API before you can implement it. There's a walkthrough of doing that in XAML controls; bind to a C++/WinRT property.

XAML applications

This scenario is about building applications and controls on the XAML UI framework. Working in a XAML application amounts to a combination of consuming and authoring. But since XAML is the dominant UI framework on Windows today, and its influence over the Windows Runtime is proportionate to that, it deserves its own category of scenario.

Be aware that XAML works best with programming languages that offer reflection. In C++/WinRT, you sometimes have to do a little extra work in order to interoperate with the XAML framework. All of those cases are covered in the documentation. Good places to start are XAML controls; bind to a C++/WinRT property and XAML custom (templated) controls with C++/WinRT.

Sample apps written in C++/WinRT

See Where can I find C++/WinRT sample apps?.

Important APIs

- SyndicationClient::RetrieveFeedAsync method
- SyndicationFeed.Items property
- winrt::hstring struct
- winrt::hresult-error struct

Related topics

- C++/CX
- Error handling with C++/WinRT
- Interop between C++/WinRT and C++/CX
- Interop between C++/WinRT and the ABI
- Move to C++/WinRT from C++/CX
- String handling in C++/WinRT

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Get Started with Win32 and C++

Article • 01/27/2022

The aim of this Get Started series is to teach you how to write a desktop program in C++ using Win32 and COM APIs.

In the first module, you'll learn step-by-step how to create and show a window. Later modules will introduce the Component Object Model (COM), graphics and text, and user input.

For this series, it is assumed that you have a good working knowledge of C++ programming. No previous experience with Windows programming is assumed. If you are new to C++, learning material is available in the C++ language documentation.

In this section

Topic	Description
Intro to Win32 programming in C++	This section describes some of the basic terminology and coding conventions used in Windows programming.
Module 1. Your First Windows Program	In this module, you will create a simple Windows program that shows a blank window.
Module 2. Using COM in Your Windows Program	This module introduces the Component Object Model (COM), which underlies many of the modern Windows APIs.
Module 3. Windows Graphics	This module introduces the Windows graphics architecture, with a focus on Direct2D.
Module 4. User Input	This module describes mouse and keyboard input.
Sample Code	Contains links to download the sample code for this series.

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

Creating an MFC Application

Article • 02/14/2023

An MFC application is an executable application for Windows that is based on the Microsoft Foundation Class (MFC) Library. MFC executables generally fall into five types: standard Windows applications, dialog boxes, forms-based applications, Explorer-style applications, and Web browser-style applications. For more information, see:

- Using the Classes to Write Windows Applications
- Creating and Displaying Dialog Boxes
- Creating a Forms-Based MFC Application
- Creating a File Explorer-Style MFC Application
- Creating a Web Browser-Style MFC Application

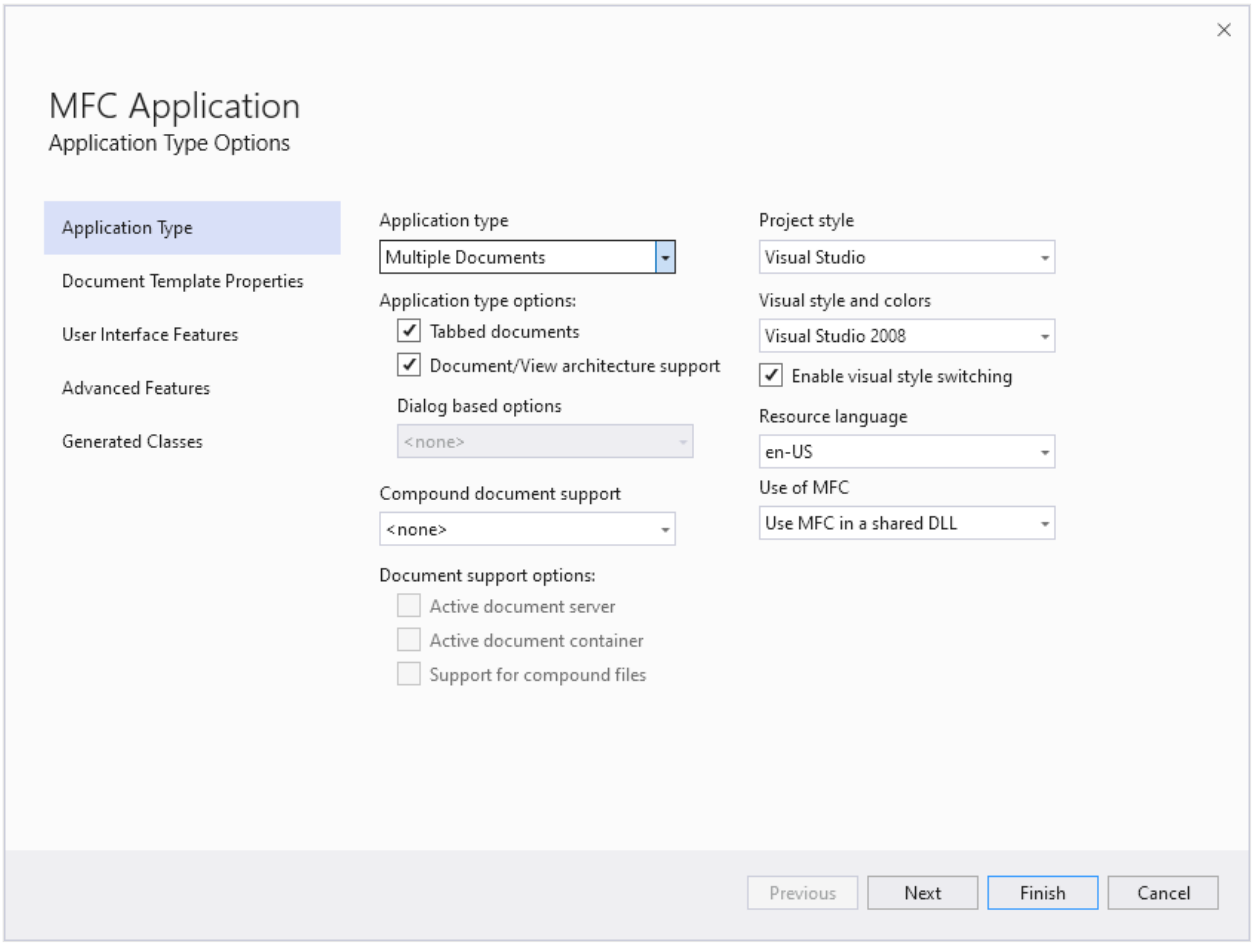
The MFC Application Wizard generates the appropriate classes and files for any of these types of applications, depending on the options you select in the wizard.

The easiest way to create an MFC application is to use the MFC Application Wizard (**MFC App project** in Visual Studio 2019). To create an MFC console application (a command-line program that uses MFC libraries but runs in the console window), use the Windows Desktop Wizard and choose the **Console Application** and **MFC Headers** options.

To create an MFC forms or dialog-based application

1. From the main menu, choose **File** > **New** > **Project**.
2. Enter "MFC" into the search box and then choose **MFC App** from the result list.
3. Modify the defaults as needed, then press **Create** to open the **MFC Application Wizard**.
4. Modify the configuration values as needed, then press **Finish**.

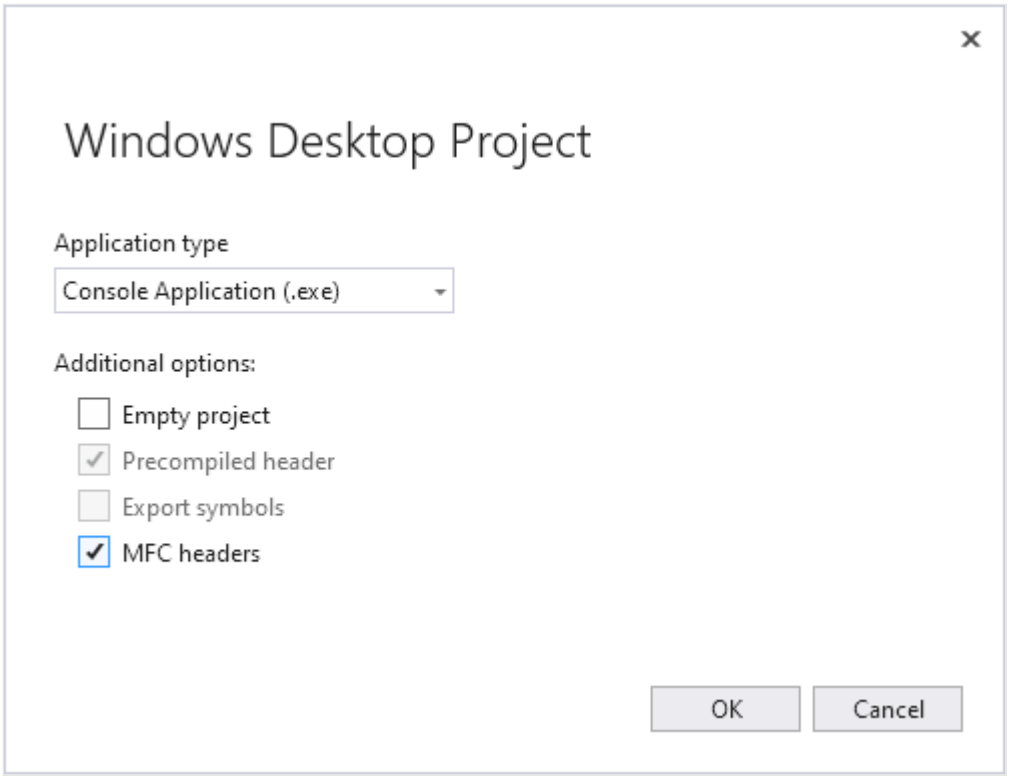
For more information, see Creating a forms-based MFC application.



To create an MFC console application

An MFC console application is a command-line program that uses MFC libraries but runs in the console window.

1. From the main menu, choose **File** > **New** > **Project**.
2. Enter "Desktop" into the search box and then choose **Windows Desktop Wizard** from the result list, then press **Next**.
3. Modify the project name and location as needed, then press **Create** to open the **Windows Desktop Wizard**.
4. Check the **MFC Headers** box and set other values as needed, then press **OK**.



Once your project is created, you can view the files created in **Solution Explorer**. For more information about the files the wizard creates for your project, see the project-generated file ReadMe.txt. For more information about the file types, see File Types Created for Visual Studio C++ projects.

See also

Adding Functionality with Code Wizards
Property Pages

Walkthrough: Create and use your own Dynamic Link Library (C++)

Article • 12/10/2021

This step-by-step walkthrough shows how to use the Visual Studio IDE to create your own dynamic link library (DLL) written in Microsoft C++ (MSVC). Then it shows how to use the DLL from another C++ app. DLLs (also known as *shared libraries* in UNIX-based operating systems) are one of the most useful kinds of Windows components. You can use them as a way to share code and resources, and to shrink the size of your apps. DLLs can even make it easier to service and extend your apps.

In this walkthrough, you'll create a DLL that implements some math functions. Then you'll create a console app that uses the functions from the DLL. You'll also get an introduction to some of the programming techniques and conventions used in Windows DLLs.

This walkthrough covers these tasks:

- Create a DLL project in Visual Studio.
- Add exported functions and variables to the DLL.
- Create a console app project in Visual Studio.
- Use the functions and variables imported from the DLL in the console app.
- Run the completed app.

Like a statically linked library, a DLL *exports* variables, functions, and resources by name. A client app *imports* the names to use those variables, functions, and resources. Unlike a statically linked library, Windows connects the imports in your app to the exports in a DLL at load time or at run time, instead of connecting them at link time. Windows requires extra information that isn't part of the standard C++ compilation model to make these connections. The MSVC compiler implements some Microsoft-specific extensions to C++ to provide this extra information. We explain these extensions as we go.

This walkthrough creates two Visual Studio solutions; one that builds the DLL, and one that builds the client app. The DLL uses the C calling convention. It can be called from apps written in other programming languages, as long as the platform, calling conventions, and linking conventions match. The client app uses *implicit linking*, where

Windows links the app to the DLL at load-time. This linking lets the app call the DLL-supplied functions just like the functions in a statically linked library.

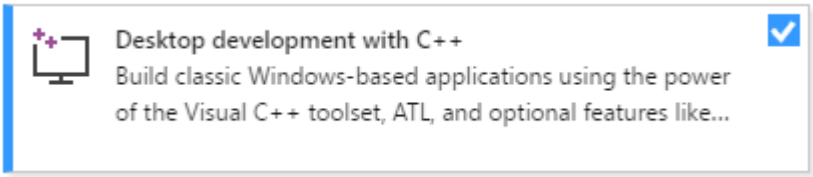
This walkthrough doesn't cover some common situations. The code doesn't show the use of C++ DLLs by other programming languages. It doesn't show how to create a resource-only DLL, or how to use explicit linking to load DLLs at run-time rather than at load-time. Rest assured, you can use MSVC and Visual Studio to do all these things.

Even though the code of the DLL is written in C++, we've used C-style interfaces for the exported functions. There are two main reasons for this: First, many other languages support imports of C-style functions. The client app doesn't have to be written in C++. Second, it avoids some common pitfalls related to exported classes and member functions. It's easy to make hard-to-diagnose errors when exporting classes, since everything referred to within a class declaration has to have an instantiation that's also exported. This restriction applies to DLLs, but not static libraries. If your classes are plain-old-data style, you shouldn't run into this issue.

For links to more information about DLLs, see Create C/C++ DLLs in Visual Studio. For more information about implicit linking and explicit linking, see Determine which linking method to use. For information about creating C++ DLLs for use with programming languages that use C-language linkage conventions, see Exporting C++ functions for use in C-language executables. For information about how to create DLLs for use with .NET languages, see Calling DLL Functions from Visual Basic Applications.

Prerequisites

- A computer that runs Microsoft Windows 7 or later versions. We recommend the latest version of Windows for the best development experience.
- A copy of Visual Studio. For information on how to download and install Visual Studio, see Install Visual Studio. When you run the installer, make sure that the **Desktop development with C++** workload is checked. Don't worry if you didn't install this workload when you installed Visual Studio. You can run the installer again and install it now.



- An understanding of the basics of using the Visual Studio IDE. If you've used Windows desktop apps before, you can probably keep up. For an introduction, see

Visual Studio IDE feature tour.

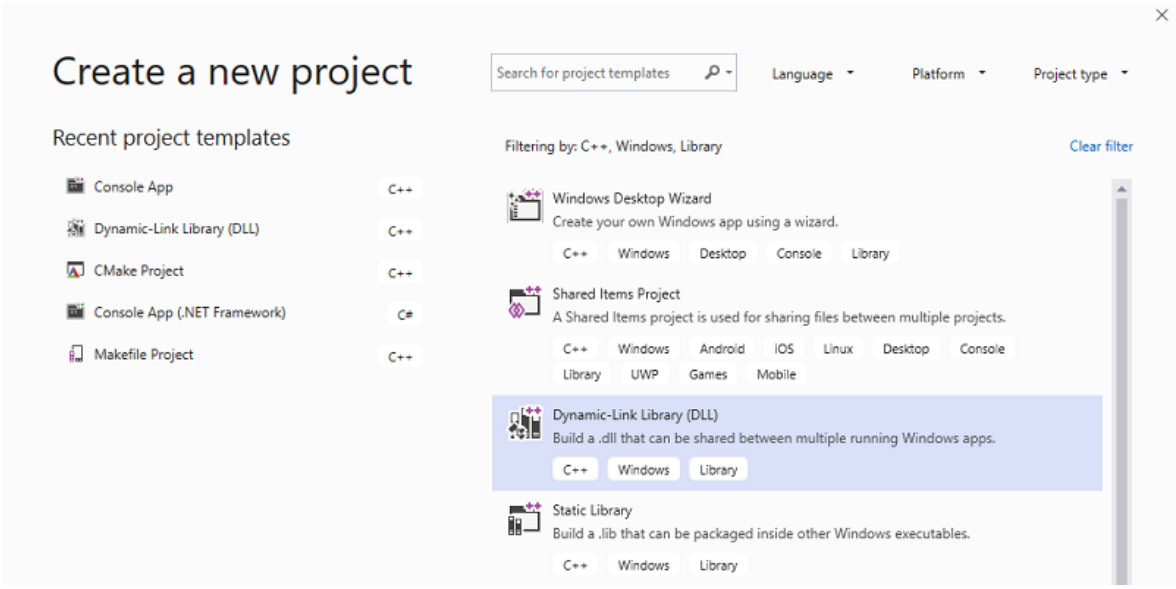
- An understanding of enough of the fundamentals of the C++ language to follow along. Don't worry, we don't do anything too complicated.

Create the DLL project

In this set of tasks, you create a project for your DLL, add code, and build it. To begin, start the Visual Studio IDE, and sign in if you need to. The instructions vary slightly depending on which version of Visual Studio you're using. Make sure you have the correct version selected in the control in the upper left of this page.

To create a DLL project in Visual Studio 2019

1. On the menu bar, choose **File** > **New** > **Project** to open the **Create a New Project** dialog box.



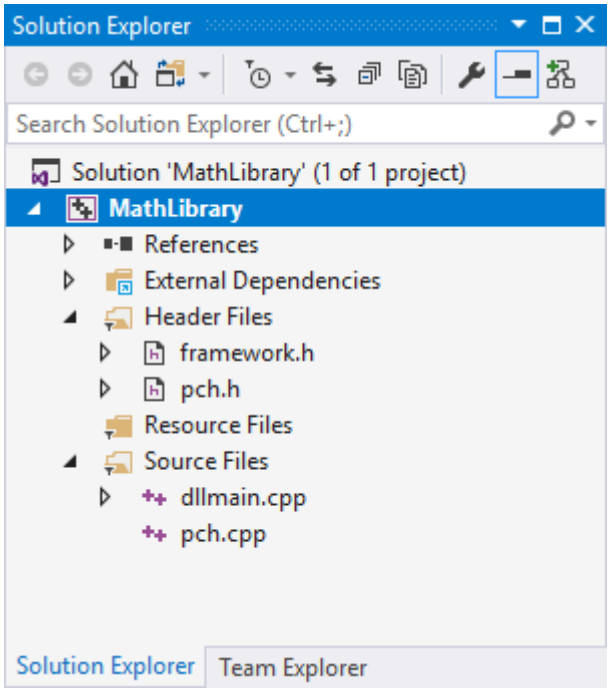
2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **Library**.

3. From the filtered list of project types, select **Dynamic-link Library (DLL)**, and then choose **Next**.

4. In the **Configure your new project** page, enter *MathLibrary* in the **Project name** box to specify a name for the project. Leave the default **Location** and **Solution name** values. Set **Solution** to **Create new solution**. Uncheck **Place solution and project in the same directory** if it's checked.

5. Choose the **Create** button to create the project.

When the solution is created, you can see the generated project and source files in the **Solution Explorer** window in Visual Studio.

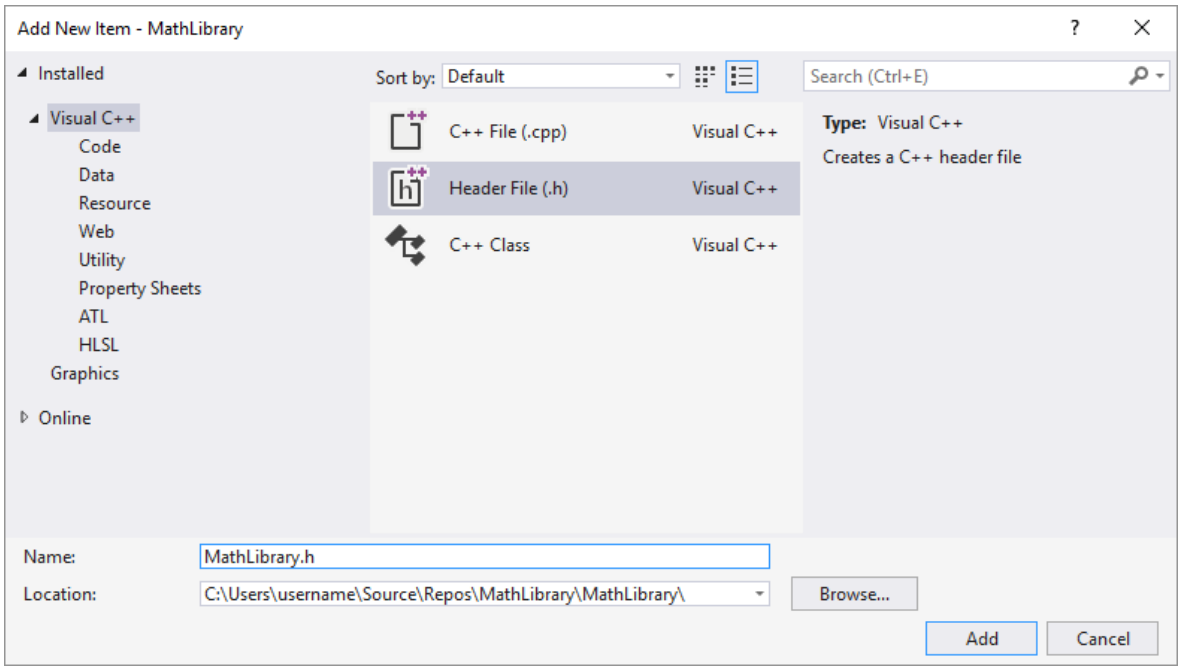


Right now, this DLL doesn't do very much. Next, you'll create a header file to declare the functions your DLL exports, and then add the function definitions to the DLL to make it more useful.

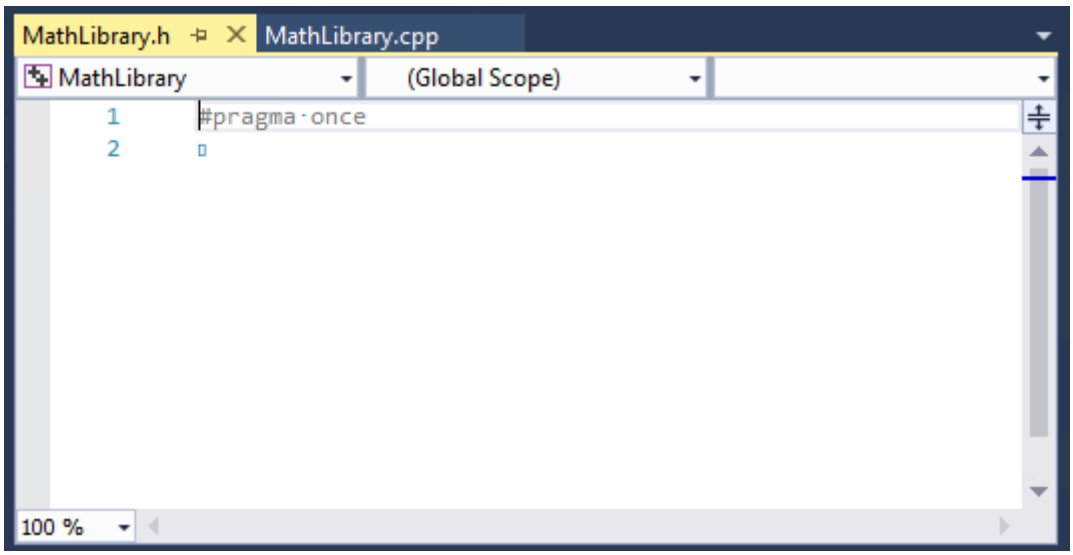
To add a header file to the DLL

1. To create a header file for your functions, on the menu bar, choose **Project** > **Add New Item**.

2. In the **Add New Item** dialog box, in the left pane, select **Visual C++**. In the center pane, select **Header File (.h)**. Specify *MathLibrary.h* as the name for the header file.



3. Choose the **Add** button to generate a blank header file, which is displayed in a new editor window.



4. Replace the contents of the header file with this code:

C++

```
// MathLibrary.h - Contains declarations of math functions
#pragma once

#ifdef MATHLIBRARY_EXPORTS
#define MATHLIBRARY_API __declspec(dllexport)
#else
#define MATHLIBRARY_API __declspec(dllimport)
#endif

// The Fibonacci recurrence relation describes a sequence F
// where F(n) is { n = 0, a
//               { n = 1, b
```

```
//               { n > 1, F(n-2) + F(n-1)
// for some initial integral values a and b.
// If the sequence is initialized F(0) = 1, F(1) = 1,
// then this relation produces the well-known Fibonacci
// sequence: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

// Initialize a Fibonacci relation sequence
// such that F(0) = a, F(1) = b.
// This function must be called before any other function.
extern "C" MATHLIBRARY_API void fibonacci_init(
    const unsigned long long a, const unsigned long long b);

// Produce the next value in the sequence.
// Returns true on success and updates current value and index;
// false on overflow, leaves current value and index unchanged.
extern "C" MATHLIBRARY_API bool fibonacci_next();

// Get the current value in the sequence.
extern "C" MATHLIBRARY_API unsigned long long fibonacci_current();

// Get the position of the current value in the sequence.
extern "C" MATHLIBRARY_API unsigned fibonacci_index();
```

This header file declares some functions to produce a generalized Fibonacci sequence, given two initial values. A call to `fibonacci_init(1, 1)` generates the familiar Fibonacci number sequence.

Notice the preprocessor statements at the top of the file. The new project template for a DLL project adds `<PROJECTNAME>_EXPORTS` to the defined preprocessor macros. In this example, Visual Studio defines `MATHLIBRARY_EXPORTS` when your MathLibrary DLL project is built.

When the `MATHLIBRARY_EXPORTS` macro is defined, the `MATHLIBRARY_API` macro sets the `__declspec(dllexport)` modifier on the function declarations. This modifier tells the compiler and linker to export a function or variable from the DLL for use by other applications. When `MATHLIBRARY_EXPORTS` is undefined, for example, when the header file is included by a client application, `MATHLIBRARY_API` applies the `__declspec(dllimport)` modifier to the declarations. This modifier optimizes the import of the function or variable in an application. For more information, see dllexport, dllimport.

To add an implementation to the DLL

1. In **Solution Explorer**, right-click on the **Source Files** node and choose **Add** > **New Item**. Create a new .cpp file called *MathLibrary.cpp*, in the same way that you added a new header file in the previous step.

2. In the editor window, select the tab for **MathLibrary.cpp** if it's already open. If not, in **Solution Explorer**, double-click **MathLibrary.cpp** in the **Source Files** folder of the **MathLibrary** project to open it.

3. In the editor, replace the contents of the MathLibrary.cpp file with the following code:

C++

```
// MathLibrary.cpp : Defines the exported functions for the DLL.
#include "pch.h" // use stdafx.h in Visual Studio 2017 and earlier
#include <utility>
#include <limits.h>
#include "MathLibrary.h"

// DLL internal state variables:
static unsigned long long previous_;  // Previous value, if any
static unsigned long long current_;   // Current sequence value
static unsigned index_;               // Current seq. position

// Initialize a Fibonacci relation sequence
// such that F(0) = a, F(1) = b.
// This function must be called before any other function.
void fibonacci_init(
    const unsigned long long a,
    const unsigned long long b)
{
    index_ = 0;
    current_ = a;
    previous_ = b; // see special case when initialized
}

// Produce the next value in the sequence.
// Returns true on success, false on overflow.
bool fibonacci_next()
{
    // check to see if we'd overflow result or position
    if ((ULLONG_MAX - previous_ < current_) ||
        (UINT_MAX == index_))
    {
        return false;
    }

    // Special case when index == 0, just return b value
    if (index_ > 0)
    {
        // otherwise, calculate next sequence value
        previous_ += current_;
    }
    std::swap(current_, previous_);
    ++index_;
    return true;
}
```

```
// Get the current value in the sequence.
unsigned long long fibonacci_current()
{
    return current_;
}

// Get the current index position in the sequence.
unsigned fibonacci_index()
{
    return index_;
}
```

To verify that everything works so far, compile the dynamic link library. To compile, choose **Build** > **Build Solution** on the menu bar. The DLL and related compiler output are placed in a folder called *Debug* directly below the solution folder. If you create a Release build, the output is placed in a folder called *Release*. The output should look something like this:

Output

```
1>------ Build started: Project: MathLibrary, Configuration: Debug Win32 ---
---
1>pch.cpp
1>dllmain.cpp
1>MathLibrary.cpp
1>Generating Code...
1>   Creating library
C:\Users\username\Source\Repos\MathLibrary\Debug\MathLibrary.lib and object
C:\Users\username\Source\Repos\MathLibrary\Debug\MathLibrary.exp
1>MathLibrary.vcxproj ->
C:\Users\username\Source\Repos\MathLibrary\Debug\MathLibrary.dll
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Congratulations, you've created a DLL using Visual Studio! Next, you'll create a client app that uses the functions exported by the DLL.

Create a client app that uses the DLL

When you create a DLL, think about how client apps may use it. To call the functions or access the data exported by a DLL, client source code must have the declarations available at compile time. At link time, the linker requires information to resolve the function calls or data accesses. A DLL supplies this information in an *import library*, a file that contains information about how to find the functions and data, instead of the actual code. And at run time, the DLL must be available to the client, in a location that the operating system can find.

Whether it's your own or from a third-party, your client app project needs several pieces of information to use a DLL. It needs to find the headers that declare the DLL exports, the import libraries for the linker, and the DLL itself. One solution is to copy all of these files into your client project. For third-party DLLs that are unlikely to change while your client is in development, this method may be the best way to use them. However, when you also build the DLL, it's better to avoid duplication. If you make a local copy of DLL files that are under development, you may accidentally change a header file in one copy but not the other, or use an out-of-date library.

To avoid out-of-sync code, we recommend you set the include path in your client project to include the DLL header files directly from your DLL project. Also, set the library path in your client project to include the DLL import libraries from the DLL project. And finally, copy the built DLL from the DLL project into your client build output directory. This step allows your client app to use the same DLL code you build.

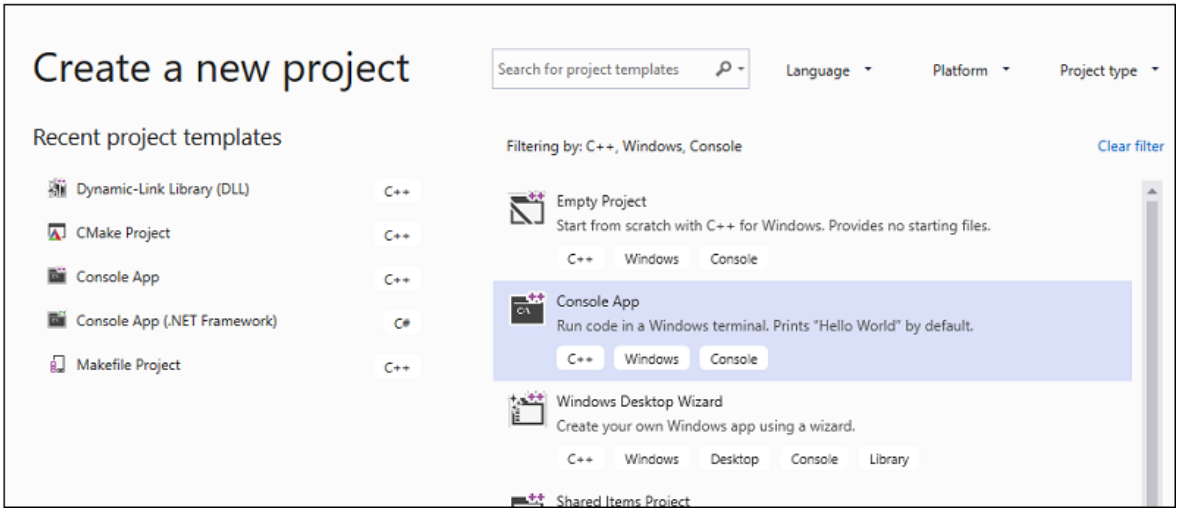
To create a client app in Visual Studio

1. On the menu bar, choose **File** > **New** > **Project** to open the **Create a new project** dialog box.

2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **Console**.

3. From the filtered list of project types, choose **Console App** then choose **Next**.

4. In the **Configure your new project** page, enter *MathClient* in the **Project name** box to specify a name for the project. Leave the default **Location** and **Solution name** values. Set **Solution** to **Create new solution**. Uncheck **Place solution and project in the same directory** if it's checked.



5. Choose the **Create** button to create the client project.

A minimal console application project is created for you. The name for the main source file is the same as the project name that you entered earlier. In this example, it's named **MathClient.cpp**. You can build it, but it doesn't use your DLL yet.

Next, to call the MathLibrary functions in your source code, your project must include the *MathLibrary.h* file. You could copy this header file into your client app project, then add it to the project as an existing item. This method can be a good choice for third-party libraries. However, if you're working on the code for your DLL and your client at the same time, the header files could get out of sync. To avoid this issue, set the **Additional Include Directories** path in your project to include the path to the original header.

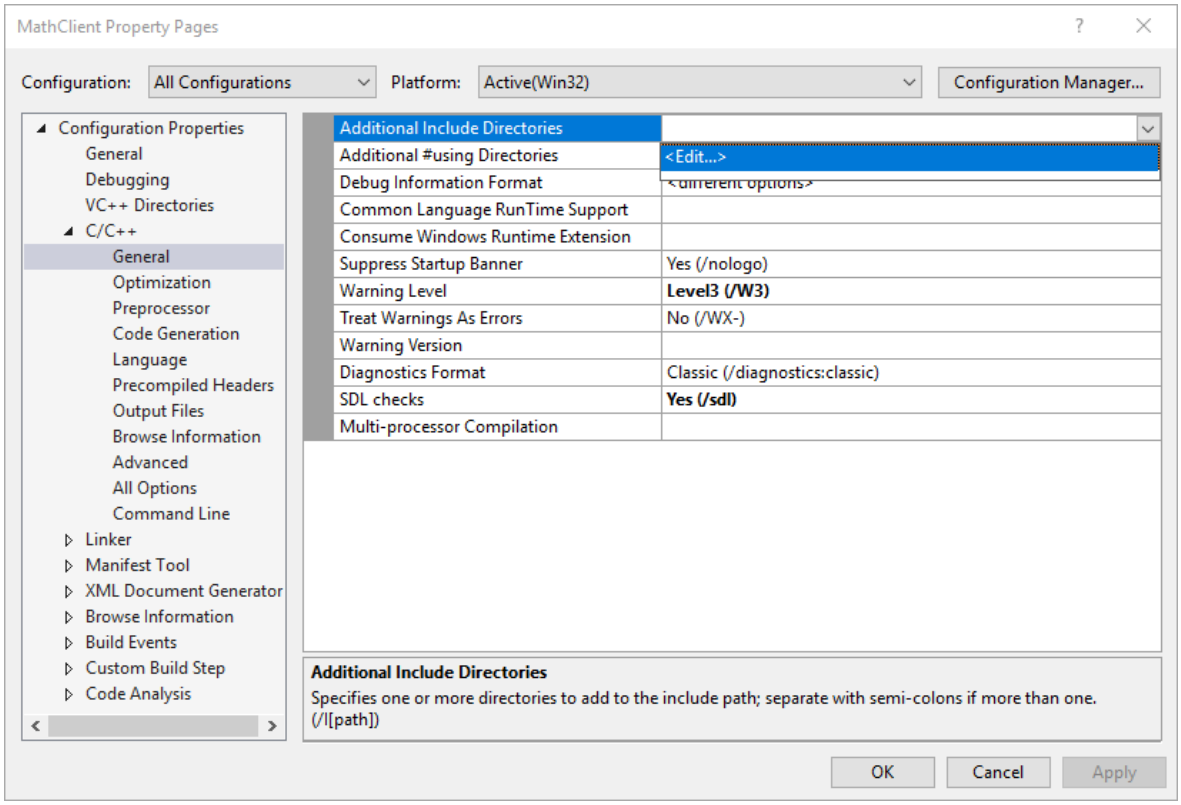
To add the DLL header to your include path

1. Right-click on the **MathClient** node in **Solution Explorer** to open the **Property Pages** dialog.

2. In the **Configuration** drop-down box, select **All Configurations** if it's not already selected.

3. In the left pane, select **Configuration Properties** > **C/C++** > **General**.

4. In the property pane, select the drop-down control next to the **Additional Include Directories** edit box, and then choose **Edit**.



5. Double-click in the top pane of the **Additional Include Directories** dialog box to enable an edit control. Or, choose the folder icon to create a new entry.

6. In the edit control, specify the path to the location of the **MathLibrary.h** header file. You can choose the ellipsis (...) control to browse to the correct folder.

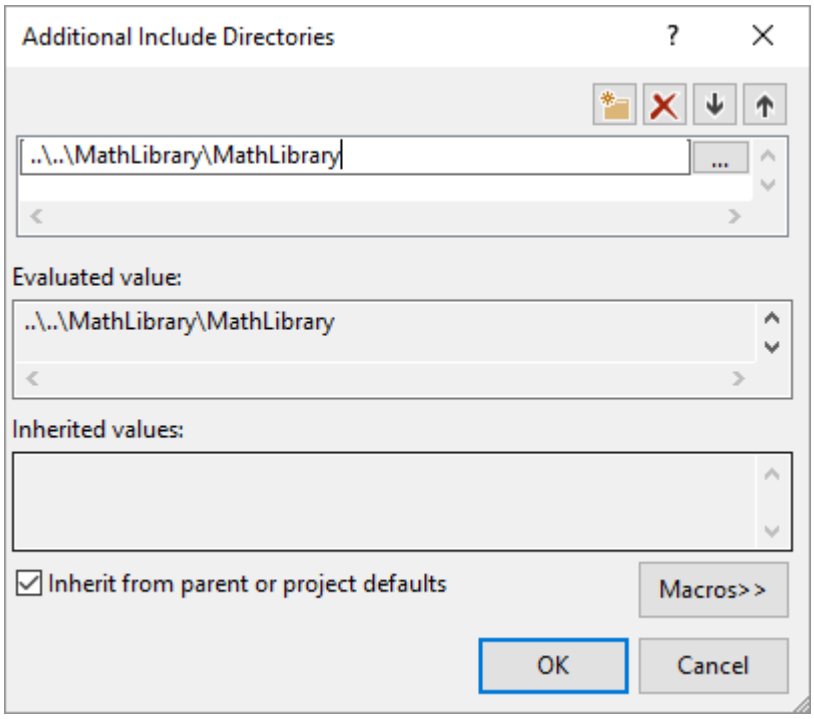
 You can also enter a relative path from your client source files to the folder that contains the DLL header files. If you followed the directions to put your client project in a separate solution from the DLL, the relative path should look like this:

   ```
   ..\..\MathLibrary\MathLibrary
   ```

 If your DLL and client projects are in the same solution, the relative path might look like this:

   ```
   ..\MathLibrary
   ```

 When the DLL and client projects are in other folders, adjust the relative path to match. Or, use the ellipsis control to browse for the folder.



7. After you've entered the path to the header file in the **Additional Include Directories** dialog box, choose the **OK** button. In the **Property Pages** dialog box, choose the **OK** button to save your changes.

You can now include the **MathLibrary.h** file and use the functions it declares in your client application. Replace the contents of **MathClient.cpp** by using this code:

```
C++
```

```
// MathClient.cpp : Client app for MathLibrary DLL.
// #include "pch.h" Uncomment for Visual Studio 2017 and earlier
#include <iostream>
#include "MathLibrary.h"

int main()
{
    // Initialize a Fibonacci relation sequence.
    fibonacci_init(1, 1);
    // Write out the sequence values until overflow.
    do {
        std::cout << fibonacci_index() << ": "
            << fibonacci_current() << std::endl;
    } while (fibonacci_next());
    // Report count of values written before overflow.
    std::cout << fibonacci_index() + 1 <<
        " Fibonacci sequence values fit in an " <<
        "unsigned 64-bit integer." << std::endl;
}
```

This code can be compiled, but not linked. If you build the client app now, the error list shows several LNK2019 errors. That's because your project is missing some information: You haven't specified that your project has a dependency on the *MathLibrary.lib* library yet. And, you haven't told the linker how to find the *MathLibrary.lib* file.

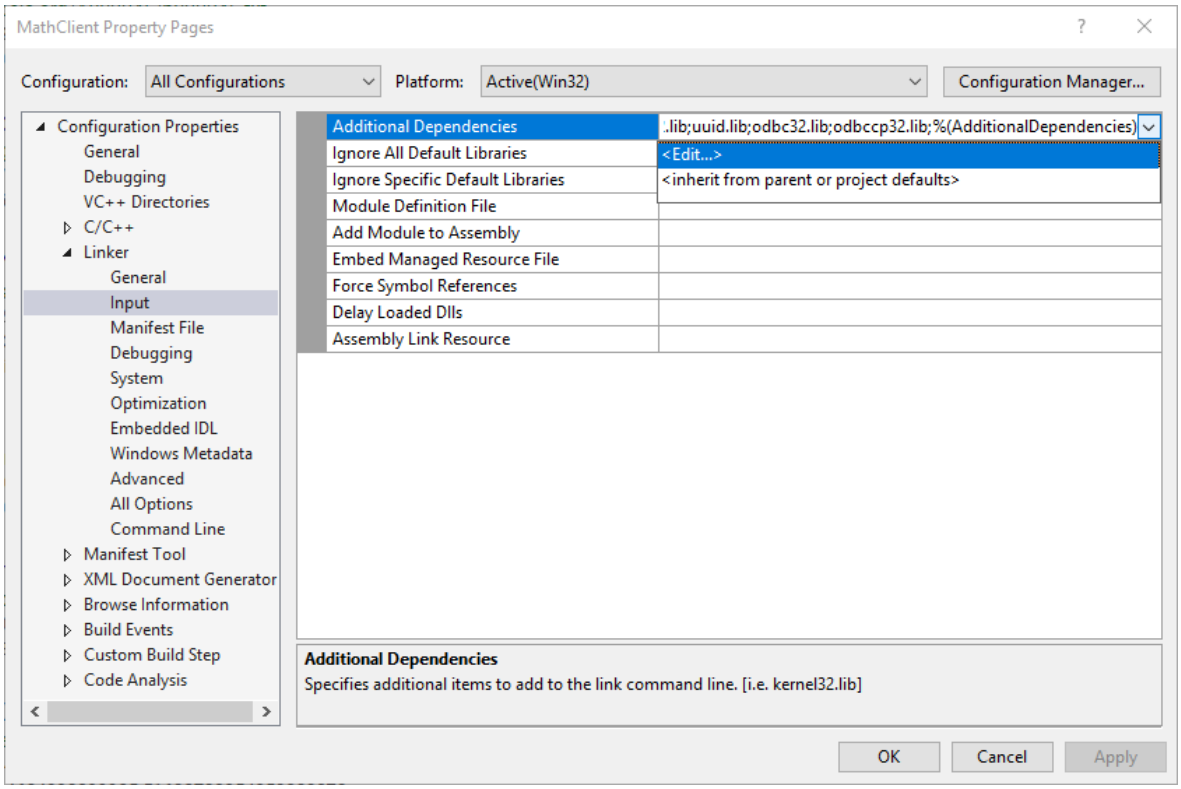
To fix this issue, you could copy the library file directly into your client app project. The linker would find and use it automatically. However, if both the library and the client app are under development, that might lead to changes in one copy that aren't shown in the other. To avoid this issue, you can set the **Additional Dependencies** property to tell the build system that your project depends on *MathLibrary.lib*. And, you can set an **Additional Library Directories** path in your project to include the path to the original library when you link.

To add the DLL import library to your project

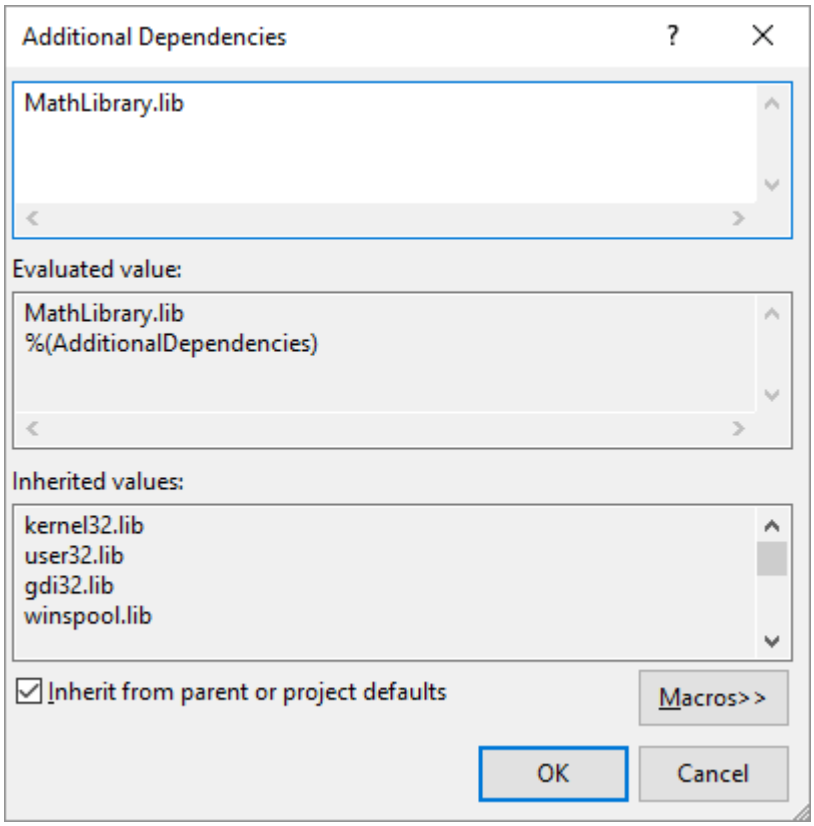
1. Right-click on the **MathClient** node in **Solution Explorer** and choose **Properties** to open the **Property Pages** dialog.

2. In the **Configuration** drop-down box, select **All Configurations** if it's not already selected. It ensures that any property changes apply to both Debug and Release builds.

3. In the left pane, select **Configuration Properties** > **Linker** > **Input**. In the property pane, select the drop-down control next to the **Additional Dependencies** edit box, and then choose **Edit**.

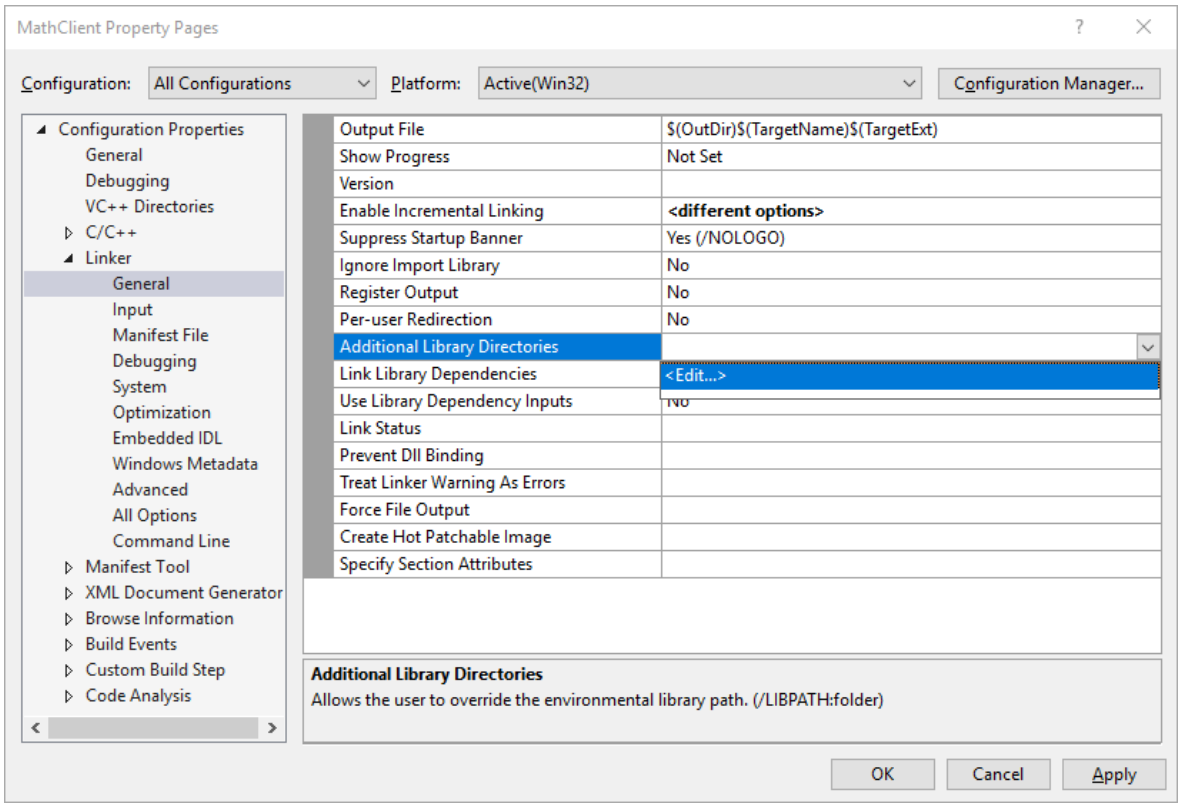


4. In the **Additional Dependencies** dialog, add *MathLibrary.lib* to the list in the top edit control.



5. Choose **OK** to go back to the **Property Pages** dialog box.

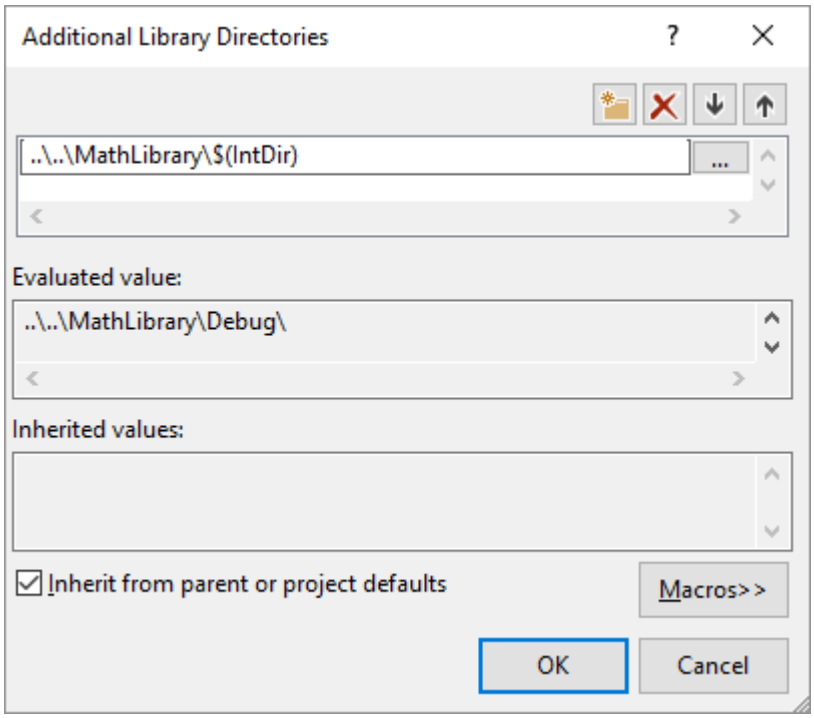
6. In the left pane, select **Configuration Properties** > **Linker** > **General**. In the property pane, select the drop-down control next to the **Additional Library Directories** edit box, and then choose **Edit**.



7. Double-click in the top pane of the **Additional Library Directories** dialog box to enable an edit control. In the edit control, specify the path to the location of the **MathLibrary.lib** file. By default, it's in a folder called *Debug* directly under the DLL solution folder. If you create a release build, the file is placed in a folder called *Release*. You can use the `$(IntDir)` macro so that the linker can find your DLL, no matter which kind of build you create. If you followed the directions to put your client project in a separate solution from the DLL project, the relative path should look like this:

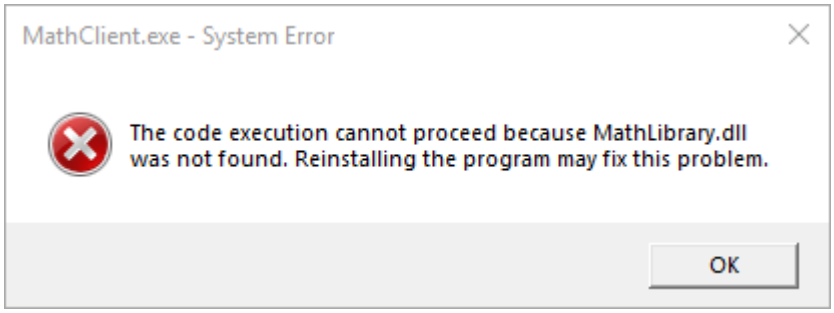
   ```
   ..\..\MathLibrary\$(IntDir)
   ```

 If your DLL and client projects are in other locations, adjust the relative path to match.



8. Once you've entered the path to the library file in the **Additional Library Directories** dialog box, choose the **OK** button to go back to the **Property Pages** dialog box. Choose **OK** to save the property changes.

Your client app can now compile and link successfully, but it still doesn't have everything it needs to run. When the operating system loads your app, it looks for the MathLibrary DLL. If it can't find the DLL in certain system directories, the environment path, or the local app directory, the load fails. Depending on the operating system, you'll see an error message like this:



One way to avoid this issue is to copy the DLL to the directory that contains your client executable as part of the build process. You can add a **Post-Build Event** to your project, to add a command that copies the DLL to your build output directory. The command specified here copies the DLL only if it's missing or has changed. It uses macros to copy to and from the Debug or Release locations, based on your build configuration.

To copy the DLL in a post-build event

1. Right-click on the **MathClient** node in **Solution Explorer** and choose **Properties** to open the **Property Pages** dialog.

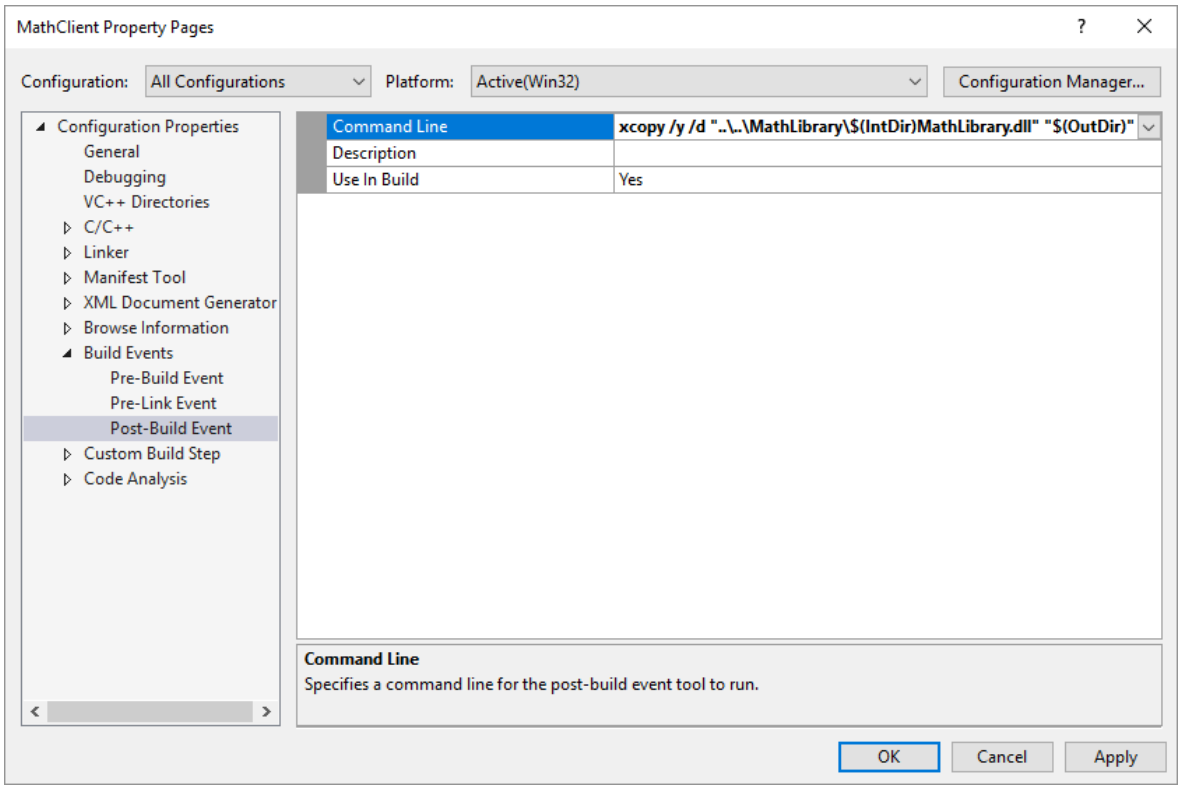
2. In the **Configuration** drop-down box, select **All Configurations** if it isn't already selected.

3. In the left pane, select **Configuration Properties** > **Build Events** > **Post-Build Event**.

4. In the property pane, select the edit control in the **Command Line** field. If you followed the directions to put your client project in a separate solution from the DLL project, then enter this command:

   ```
   xcopy /y /d "..\..\MathLibrary\$(IntDir)MathLibrary.dll" "$(OutDir)"
   ```

 If your DLL and client projects are in other directories, change the relative path to the DLL to match.



5. Choose the **OK** button to save your changes to the project properties.

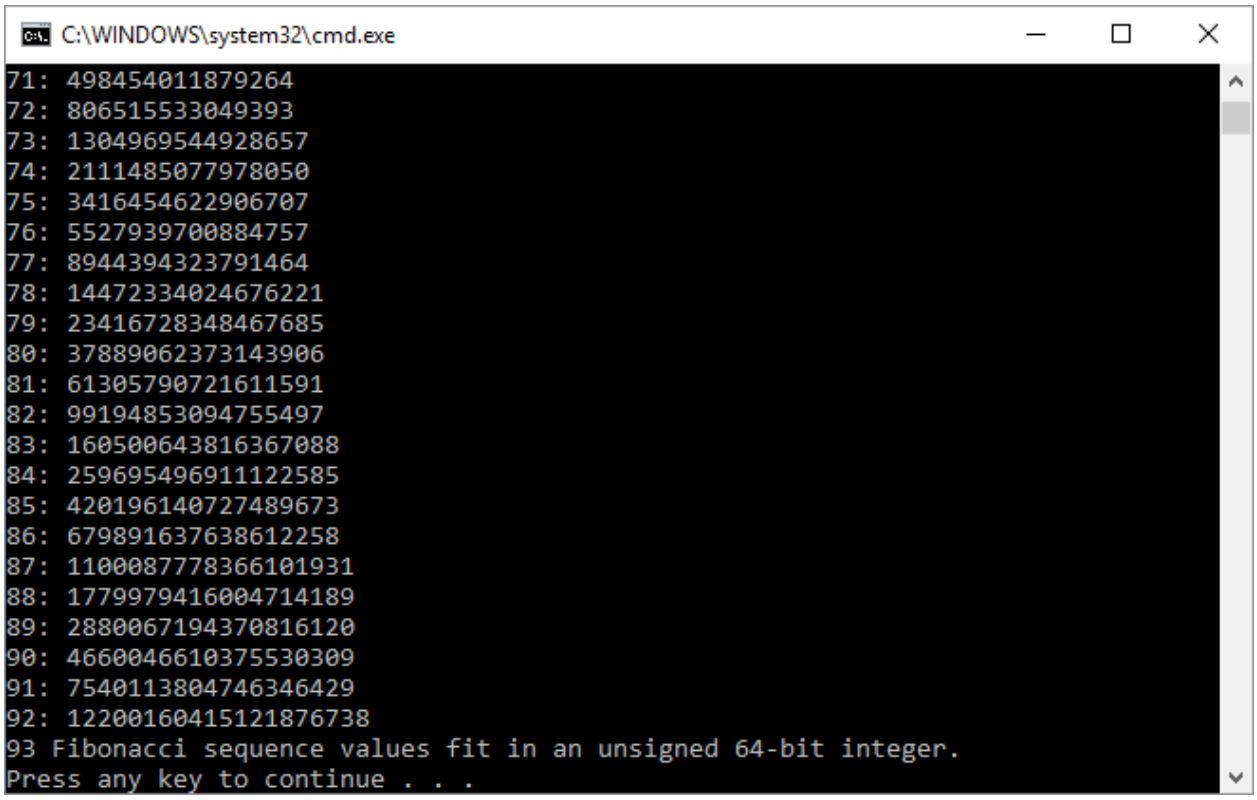
Now your client app has everything it needs to build and run. Build the application by choosing **Build** > **Build Solution** on the menu bar. The **Output** window in Visual Studio should have something like the following example depending on your version of Visual Studio:

Output

```
1>------ Build started: Project: MathClient, Configuration: Debug Win32 ----
--
1>MathClient.cpp
1>MathClient.vcxproj ->
```

```
C:\Users\username\Source\Repos\MathClient\Debug\MathClient.exe
1>1 File(s) copied
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
```

Congratulations, you've created an application that calls functions in your DLL. Now run your application to see what it does. On the menu bar, choose **Debug** > **Start Without Debugging**. Visual Studio opens a command window for the program to run in. The last part of the output should look like:



Press any key to dismiss the command window.

Now that you've created a DLL and a client application, you can experiment. Try setting breakpoints in the code of the client app, and run the app in the debugger. See what happens when you step into a library call. Add other functions to the library, or write another client app that uses your DLL.

When you deploy your app, you must also deploy the DLLs it uses. The simplest way to make the DLLs that you build, or that you include from third parties, available is to put them in the same directory as your app. It's known as *app-local deployment*. For more information about deployment, see Deployment in Visual C++.

See also

Calling DLL Functions from Visual Basic Applications

Walkthrough: Create and use a static library

Article • 10/29/2021

This step-by-step walkthrough shows how to create a static library (.lib file) for use with C++ apps. Using a static library is a great way to reuse code. Rather than reimplementing the same routines in every app that requires the functionality, you write them one time in a static library and then reference it from the apps. Code linked from a static library becomes part of your app—you don't have to install another file to use the code.

This walkthrough covers these tasks:

- Create a static library project
- Add a class to the static library
- Create a C++ console app that references the static library
- Use the functionality from the static library in the app
- Run the app

Prerequisites

An understanding of the fundamentals of the C++ language.

Create a static library project

The instructions for how to create the project vary depending on your version of Visual Studio. To see the documentation for your preferred version of Visual Studio, use the **Version** selector control. It's found at the top of the table of contents on this page.

To create a static library project in Visual Studio

1. On the menu bar, choose **File** > **New** > **Project** to open the **Create a New Project** dialog.
2. At the top of the dialog, set **Language** to **C++**, set **Platform** to **Windows**, and set **Project type** to **Library**.

3. From the filtered list of project types, select **Windows Desktop Wizard**, then choose **Next**.

4. In the **Configure your new project** page, enter *MathLibrary* in the **Project name** box to specify a name for the project. Enter *StaticMath* in the **Solution name** box. Choose the **Create** button to open the **Windows Desktop Project** dialog.

5. In the **Windows Desktop Project** dialog, under **Application type**, select **Static Library (.lib)**.

6. Under **Additional options**, uncheck the **Precompiled header** check box if it's checked. Check the **Empty project** box.

7. Choose **OK** to create the project.

Add a class to the static library

To add a class to the static library

1. To create a header file for a new class, right-click to open the shortcut menu for the **MathLibrary** project in **Solution Explorer**, and then choose **Add** > **New Item**.

2. In the **Add New Item** dialog box, select **Visual C++** > **Code**. In the center pane, select **Header File (.h)**. Specify a name for the header file—for example, *MathLibrary.h*—and then choose the **Add** button. A nearly blank header file is displayed.

3. Add a declaration for a class named `Arithmetic` to do common mathematical operations such as addition, subtraction, multiplication, and division. The code should resemble:

```
C++
```

```
// MathLibrary.h
#pragma once

namespace MathLibrary
{
    class Arithmetic
    {
    public:
        // Returns a + b
        static double Add(double a, double b);

        // Returns a - b
        static double Subtract(double a, double b);
```

```
        // Returns a * b
        static double Multiply(double a, double b);

        // Returns a / b
        static double Divide(double a, double b);
    };
}
```

4. To create a source file for the new class, open the shortcut menu for the **MathLibrary** project in **Solution Explorer**, and then choose **Add** > **New Item**.

5. In the **Add New Item** dialog box, in the center pane, select **C++ File (.cpp)**. Specify a name for the source file—for example, *MathLibrary.cpp*—and then choose the **Add** button. A blank source file is displayed.

6. Use this source file to implement the functionality for class `Arithmetic`. The code should resemble:

C++

```
// MathLibrary.cpp
// compile with: cl /c /EHsc MathLibrary.cpp
// post-build command: lib MathLibrary.obj

#include "MathLibrary.h"

namespace MathLibrary
{
    double Arithmetic::Add(double a, double b)
    {
        return a + b;
    }

    double Arithmetic::Subtract(double a, double b)
    {
        return a - b;
    }

    double Arithmetic::Multiply(double a, double b)
    {
        return a * b;
    }

    double Arithmetic::Divide(double a, double b)
    {
        return a / b;
    }
}
```

7. To build the static library, select **Build** > **Build Solution** on the menu bar. The build creates a static library, *MathLibrary.lib*, that can be used by other programs.

> **Note**
>
> When you build on the Visual Studio command line, you must build the program in two steps. First, run `cl /c /EHsc MathLibrary.cpp` to compile the code and create an object file that's named *MathLibrary.obj*. (The `cl` command invokes the compiler, Cl.exe, and the `/c` option specifies compile without linking. For more information, see **/c (Compile Without Linking)**.) Second, run `lib MathLibrary.obj` to link the code and create the static library *MathLibrary.lib*. (The `lib` command invokes the Library Manager, Lib.exe. For more information, see **LIB Reference**.)

Create a C++ console app that references the static library

To create a C++ console app that references the static library in Visual Studio

1. In **Solution Explorer**, right-click on the top node, **Solution 'StaticMath'**, to open the shortcut menu. Choose **Add** > **New Project** to open the **Add a New Project** dialog.

2. At the top of the dialog, set the **Project type** filter to **Console**.

3. From the filtered list of project types, choose **Console App** then choose **Next**. In the next page, enter *MathClient* in the **Name** box to specify a name for the project.

4. Choose the **Create** button to create the client project.

5. After you create a console app, an empty program is created for you. The name for the source file is the same as the name that you chose earlier. In the example, it's named `MathClient.cpp`.

Use the functionality from the static library in the app

To use the functionality from the static library in the app

1. Before you can use the math routines in the static library, you must reference it. Open the shortcut menu for the **MathClient** project in **Solution Explorer**, and then choose **Add** > **Reference**.

2. The **Add Reference** dialog box lists the libraries that you can reference. The **Projects** tab lists the projects in the current solution and any libraries they reference. Open the **Projects** tab, select the **MathLibrary** check box, and then choose the **OK** button.

3. To reference the `MathLibrary.h` header file, you must modify the included directories path. In **Solution Explorer**, right-click on **MathClient** to open the shortcut menu. Choose **Properties** to open the **MathClient Property Pages** dialog box.

4. In the **MathClient Property Pages** dialog box, set the **Configuration** drop-down to **All Configurations**. Set the **Platform** drop-down to **All Platforms**.

5. Select the **Configuration Properties** > **C/C++** > **General** property page. In the **Additional Include Directories** property, specify the path of the **MathLibrary** directory, or browse for it.

 To browse for the directory path:

 a. Open the **Additional Include Directories** property value drop-down list, and then choose **Edit**.

 b. In the **Additional Include Directories** dialog box, double-click in the top of the text box. Then choose the ellipsis button (**...**) at the end of the line.

 c. In the **Select Directory** dialog box, navigate up a level, and then select the **MathLibrary** directory. Then choose the **Select Folder** button to save your selection.

 d. In the **Additional Include Directories** dialog box, choose the **OK** button.

 e. In the **Property Pages** dialog box, choose the **OK** button to save your changes to the project.

6. You can now use the `Arithmetic` class in this app by including the `#include "MathLibrary.h"` header in your code. Replace the contents of `MathClient.cpp` with this code:

   ```
   C++
   ```

```
// MathClient.cpp
// compile with: cl /EHsc MathClient.cpp /link MathLibrary.lib

#include <iostream>
#include "MathLibrary.h"

int main()
{
    double a = 7.4;
    int b = 99;

    std::cout << "a + b = " <<
        MathLibrary::Arithmetic::Add(a, b) << std::endl;
    std::cout << "a - b = " <<
        MathLibrary::Arithmetic::Subtract(a, b) << std::endl;
    std::cout << "a * b = " <<
        MathLibrary::Arithmetic::Multiply(a, b) << std::endl;
    std::cout << "a / b = " <<
        MathLibrary::Arithmetic::Divide(a, b) << std::endl;

    return 0;
}
```

7. To build the executable, choose **Build** > **Build Solution** on the menu bar.

Run the app

To run the app

1. Make sure that **MathClient** is selected as the default project. To select it, right-click to open the shortcut menu for **MathClient** in **Solution Explorer**, and then choose **Set as StartUp Project**.

2. To run the project, on the menu bar, choose **Debug** > **Start Without Debugging**. The output should resemble:

Output

```
a + b = 106.4
a - b = -91.6
a * b = 732.6
a / b = 0.0747475
```

See also

Walkthrough: Creating and Using a Dynamic Link Library (C++)

Desktop Applications (Visual C++)

Walkthrough: Compile a C++/CLI program that targets the CLR in Visual Studio

Article • 10/29/2021

By using C++/CLI you can create C++ programs that use .NET classes as well as native C++ types. C++/CLI is intended for use in console applications and in DLLs that wrap native C++ code and make it accessible from .NET programs. To create a Windows user interface based on .NET, use C# or Visual Basic.

For this procedure, you can type your own C++ program or use one of the sample programs. The sample program that we use in this procedure creates a text file named textfile.txt, and saves it to the project directory.

Prerequisites

- An understanding of the fundamentals of the C++ language.
- In Visual Studio 2017 and later, C++/CLI support is an optional component. To install it, open the **Visual Studio Installer** from the Windows Start menu. Make sure that the **Desktop development with C++** tile is checked, and in the **Optional** components section, also check **C++/CLI Support**.

Create a new project

The following steps vary depending on which version of Visual Studio you are using. To see the documentation for your preferred version of Visual Studio, use the **Version** selector control. It's found at the top of the table of contents on this page.

To create a C++/CLI project in Visual Studio

1. In **Solution Explorer**, right-click on the top to open the **Create a New Project** dialog box.

2. At the top of the dialog, type **CLR** in the search box and then choose **CLR Empty Project** from the results list.

3. Choose the **Create** button to create the project.

Add a source file

1. If **Solution Explorer** isn't visible, click **Solution Explorer** on the **View** menu.

2. Add a new source file to the project:

 - Right-click the **Source Files** folder in **Solution Explorer**, point to **Add**, and click **New Item**.

 - Click **C++ File (.cpp)** and type a file name and then click **Add**.

 The **.cpp** file appears in the **Source Files** folder in **Solution Explorer** and a tabbed window appears where you type the code you want in that file.

3. Click in the newly created tab in Visual Studio and type a valid Visual C++ program, or copy and paste one of the sample programs.

 For example, you can use the How to: Write a Text File (C++/CLI) sample program (in the **File Handling and I/O** node of the Programming Guide).

 If you use the sample program, notice that you use the `gcnew` keyword instead of `new` when creating a .NET object, and that `gcnew` returns a handle (`^`) rather than a pointer (`*`):

   ```
   StreamWriter^ sw = gcnew StreamWriter(fileName);
   ```

 For more information on C++/CLI syntax, see Component Extensions for Runtime Platforms.

4. On the **Build** menu, click **Build Solution**.

 The **Output** window displays information about the compilation progress, such as the location of the build log and a message that indicates the build status.

 If you make changes and run the program without doing a build, a dialog box might indicate that the project is out of date. Select the checkbox on this dialog before you click **OK** if you want Visual Studio to always use the current versions of files instead of prompting you each time it builds the application.

5. On the **Debug** menu, click **Start without Debugging**.

6. If you used the sample program, when you run the program a command window is displayed that indicates the text file has been created.

 The **textfile.txt** text file is now located in your project directory. You can open this file by using Notepad.

> **Note**
>
> Choosing the empty CLR project template automatically set the `/clr` compiler option. To verify this, right-click the project in **Solution Explorer** and clicking **Properties**, and then check the **Common Language Runtime support** option in the **General** node of **Configuration Properties**.

See also

C++ Language Reference
Projects and build systems

Create a simple Universal Windows Platform (UWP) game with DirectX

Article • 10/20/2022

In this set of tutorials, you'll learn how to use DirectX and C++/WinRT to create the basic Universal Windows Platform (UWP) sample game named **Simple3DGameDX**. The gameplay takes place in a simple first-person 3D shooting gallery.

> **Note**
>
> The link from which you can download the **Simple3DGameDX** sample game itself is **Direct3D sample game**. The C++/WinRT source code is in the folder named `cppwinrt`. For info about other UWP sample apps, see **Sample applications for Windows development**.

These tutorials cover all of the major parts of a game, including the processes for loading assets such as arts and meshes, creating a main game loop, implementing a simple rendering pipeline, and adding sound and controls.

You'll also see UWP game development techniques and considerations. We'll focus on key UWP DirectX game development concepts, and call out Windows-Runtime-specific considerations around those concepts.

Objective

To learn about the basic concepts and components of a UWP DirectX game, and to become more comfortable designing UWP games with DirectX.

What you need to know

For this tutorial, you need to be familiar with these subjects.

- C++/WinRT. C++/WinRT is a standard modern C++17 language projection for Windows APIs, implemented as a header-file-based library, and designed to provide you with first-class access to the modern Windows APIs.
- Basic linear algebra and Newtonian physics concepts.
- Basic graphics programming terminology.
- Basic Windows programming concepts.
- Basic familiarity with the Direct2D and Direct3D 11 APIs.

Direct3D UWP shooting gallery sample

The **Simple3DGameDX** sample game implements a simple first-person 3D shooting gallery, where the player fires balls at moving targets. Hitting each target awards a set number of points, and the player can progress through 6 levels of increasing challenge. At the end of the levels, the points are tallied, and the player is awarded a final score.

The sample demonstrates these game concepts.

- Interoperation between DirectX 11.1 and the Windows Runtime
- A first-person 3D perspective and camera
- Stereoscopic 3D effects
- Collision-detection between objects in 3D
- Handling player input for mouse, touch, and Xbox controller controls
- Audio mixing and playback
- A basic game state-machine



Topic	Description
Set up the game project	The first step in developing your game is to set up a project in Microsoft Visual Studio. After you've configured a project specifically for game development, you could later re-use it as a kind of template.
Define the game's UWP app framework	The first step in coding a Universal Windows Platform (UWP) game is building the framework that lets the app object interact with Windows.
Game flow management	Define the high-level state machine to enable player and system interaction. Learn how UI interacts with the overall game's state machine and how to create event handlers for UWP games.

Topic	Description
Define the main game object	Now, we look at the details of the sample game's main object and how the rules it implements translate into interactions with the game world.
Rendering framework I: Intro to rendering	Learn how to develop the rendering pipeline to display graphics. Intro to rendering.
Rendering framework II: Game rendering	Learn how to assemble the rendering pipeline to display graphics. Game rendering, set up and prepare data.
Add a user interface	Learn how to add a 2D user interface overlay to a DirectX UWP game.
Add controls	Now, we take a look at how the sample game implements move-look controls in a 3-D game, and how to develop basic touch, mouse, and game controller controls.
Add sound	Develop a simple sound engine using XAudio2 APIs to playback game music and sound effects.
Extend the sample game	Learn how to implement a XAML overlay for a UWP DirectX game.

Tutorial: Open a project from a repo

Article • 12/19/2024

In this tutorial, you use Visual Studio to connect to a repository, or *repo*, for the first time, clone it, and then open a project from it.

In this tutorial, you learn how to:

- ✓ Open a project from a GitHub repo
- ✓ Browse to an Azure DevOps repo

Prerequisites

- If you don't have Visual Studio yet, go to Visual Studio downloads to install it for free.

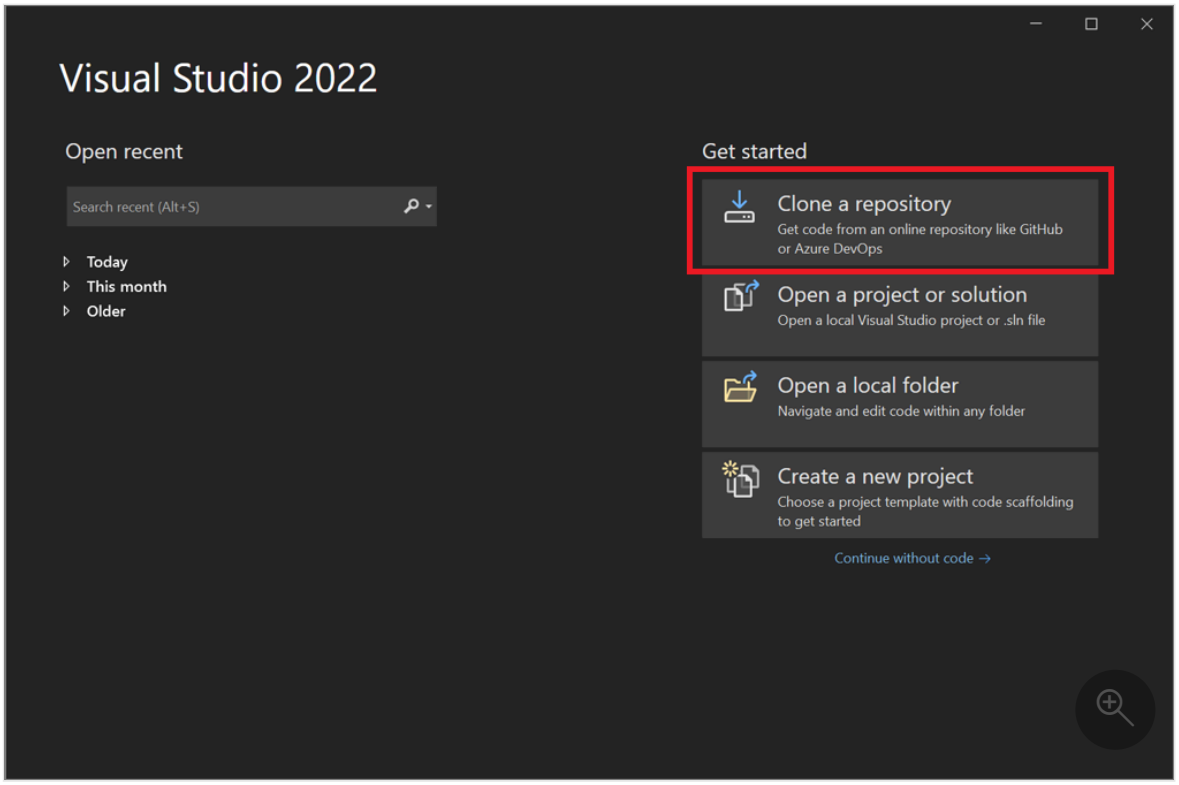
Open a project from a GitHub repo

Visual Studio makes it easy to open a project from a repo. You can do so when you start Visual Studio, or you can do so directly from within the Visual Studio IDE.

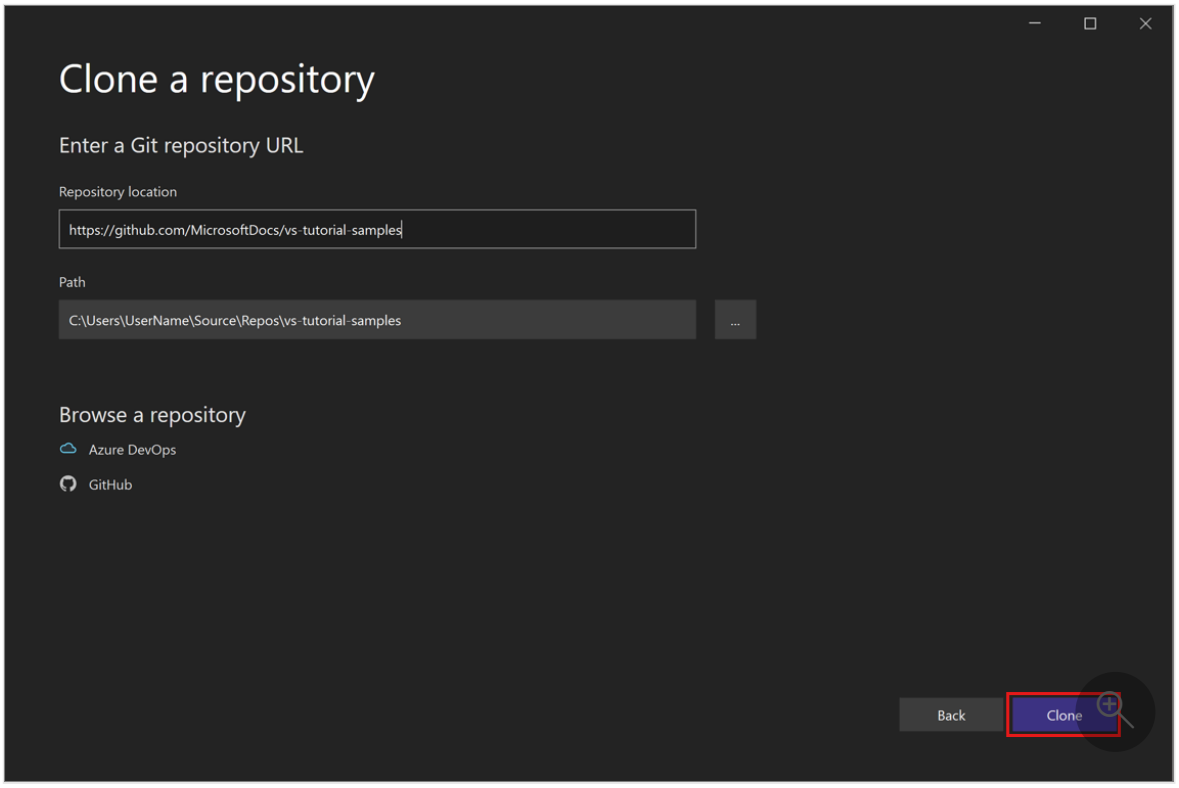
Here's how.

Use the start window

1. Open Visual Studio.
2. On the start window, select **Clone a repository**.



3. Enter or type the repository location, and then select **Clone**.



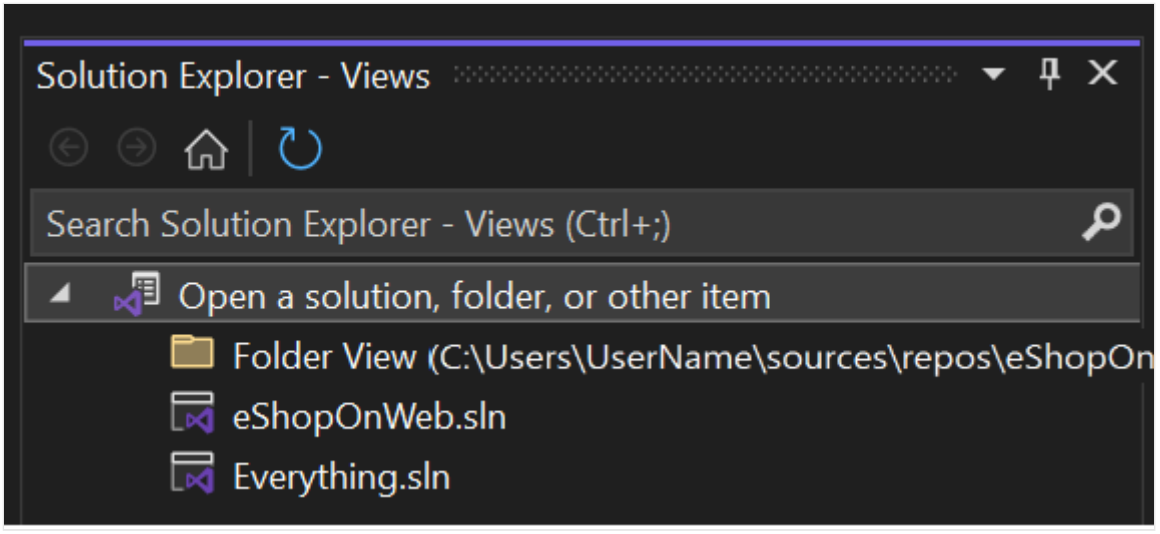
4. If you're not already signed in, you might be prompted to sign into Visual Studio or your GitHub account.

> **Tip**
>
> For more information about signing in to Visual Studio, see Sign in or switch Visual Studio user accounts. For specific information about how to use your

GitHub account to sign in, see Add your GitHub accounts to your Visual Studio keychain. You might receive a trust notification. For more information, see Configure trust settings for files and folders.

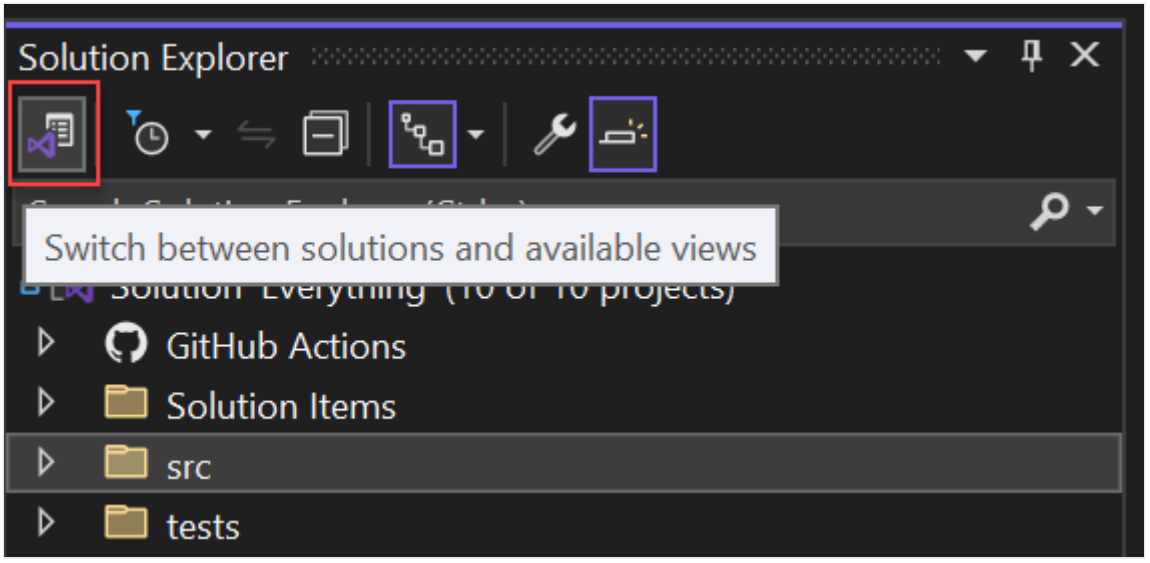
View files in Solution Explorer

Visual Studio loads the solutions from the repository by using the **Folder View** in Solution Explorer.



You can view a solution in **Solution View** by double-clicking its `.sln` file.

You can select **Switch Views** to switch between folder view and solution view.



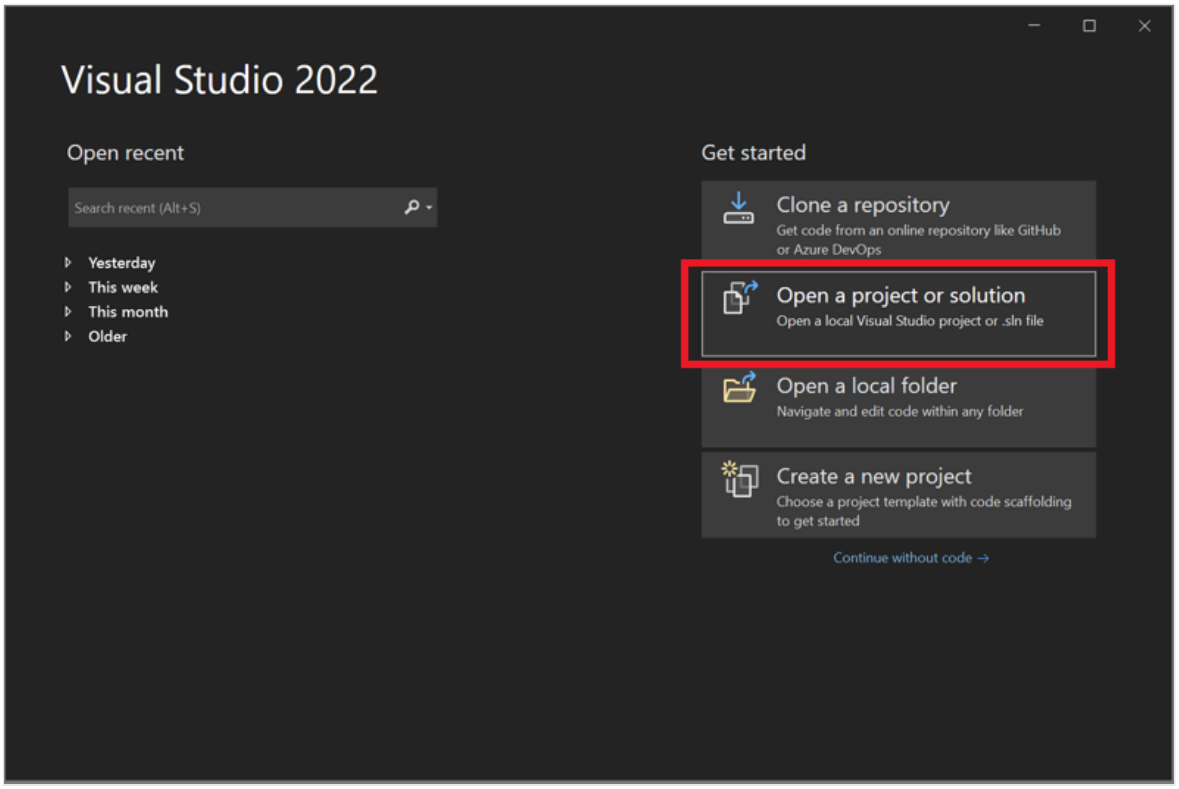
Tip

You can change from the default Folder View to Solution View from the **Git** menu. Select **Settings** > **Source Control** > **Git Global Settings** > **Automatically load the solution when opening a Git repository**.

Open a project locally from a previously cloned GitHub repo

1. Open Visual Studio.
2. On the start window, select **Open a project or solution**.

 Visual Studio opens an instance of File Explorer, where you can browse to your solution or project, and then select it to open it.



> **Tip**
>
> If you opened the project or solution recently, select it from the **Open recent** section.

Start coding!

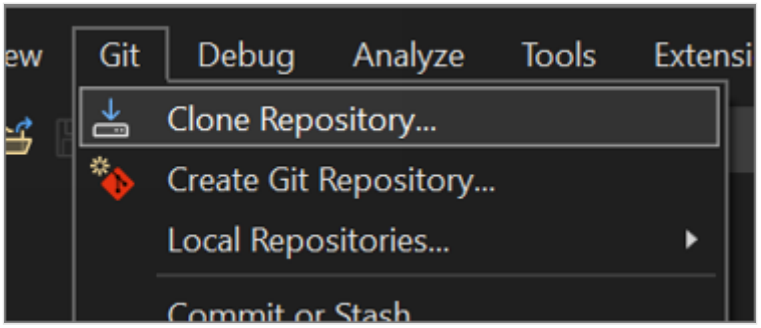
Use the IDE

You can also use the **Git** menu or the **Select Repository** control in the Visual Studio IDE to interact with a repository's folders and files.

Here's how.

To clone a repo and open a project

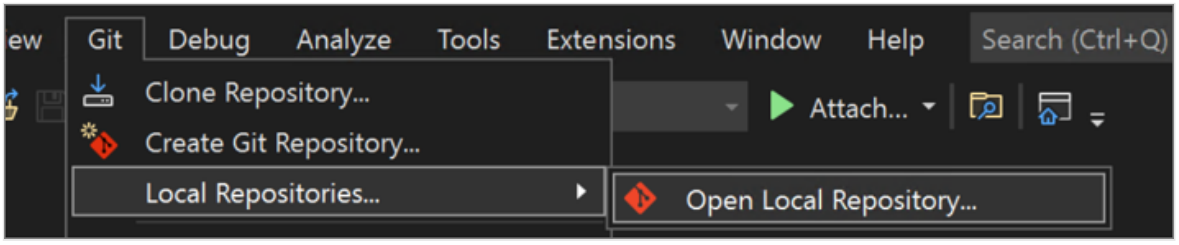
1. In the Visual Studio IDE, select the **Git** menu, and then select **Clone Repository**.



2. Follow the prompts to connect to the Git repository that includes the files that you're looking for.

To open local folders and files

1. In the Visual Studio IDE, select the **Git** menu, select **Local Repositories**, and then select **Open Local Repository**.



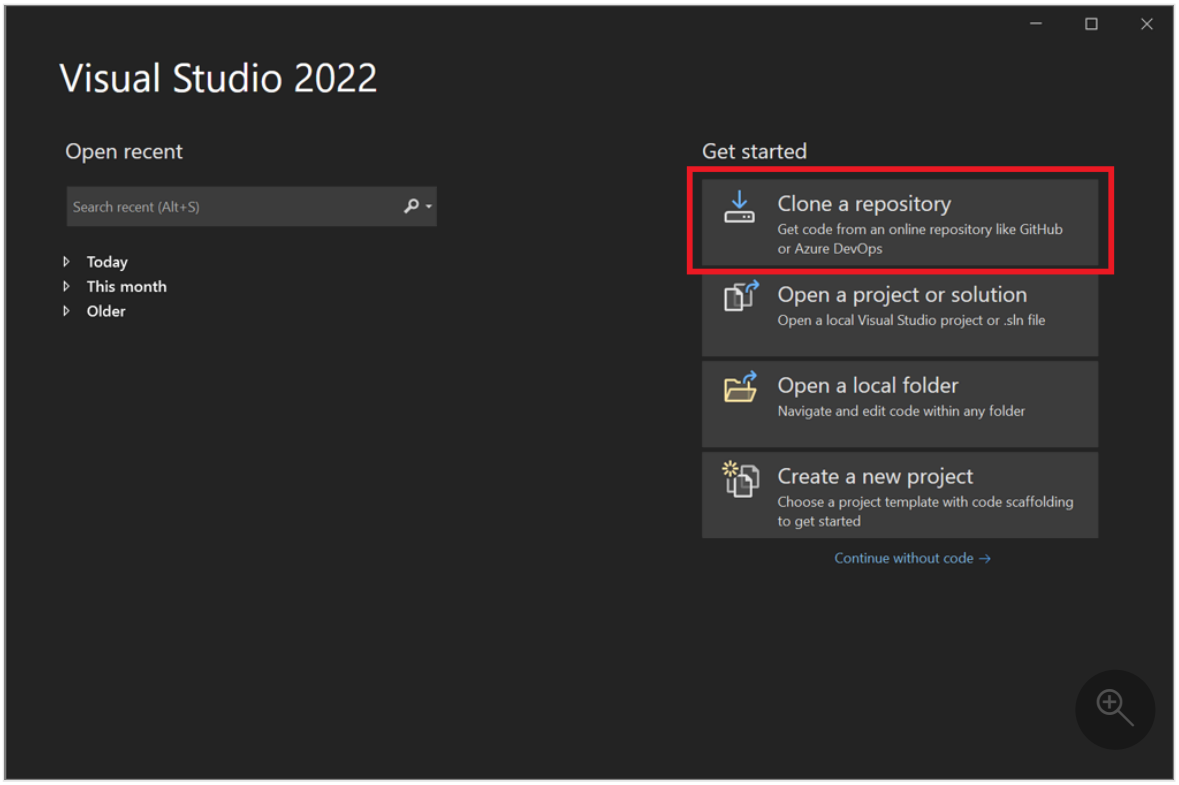
2. Follow the prompts to connect to the Git repository that has the files that you're looking for.

Browse to an Azure DevOps repo

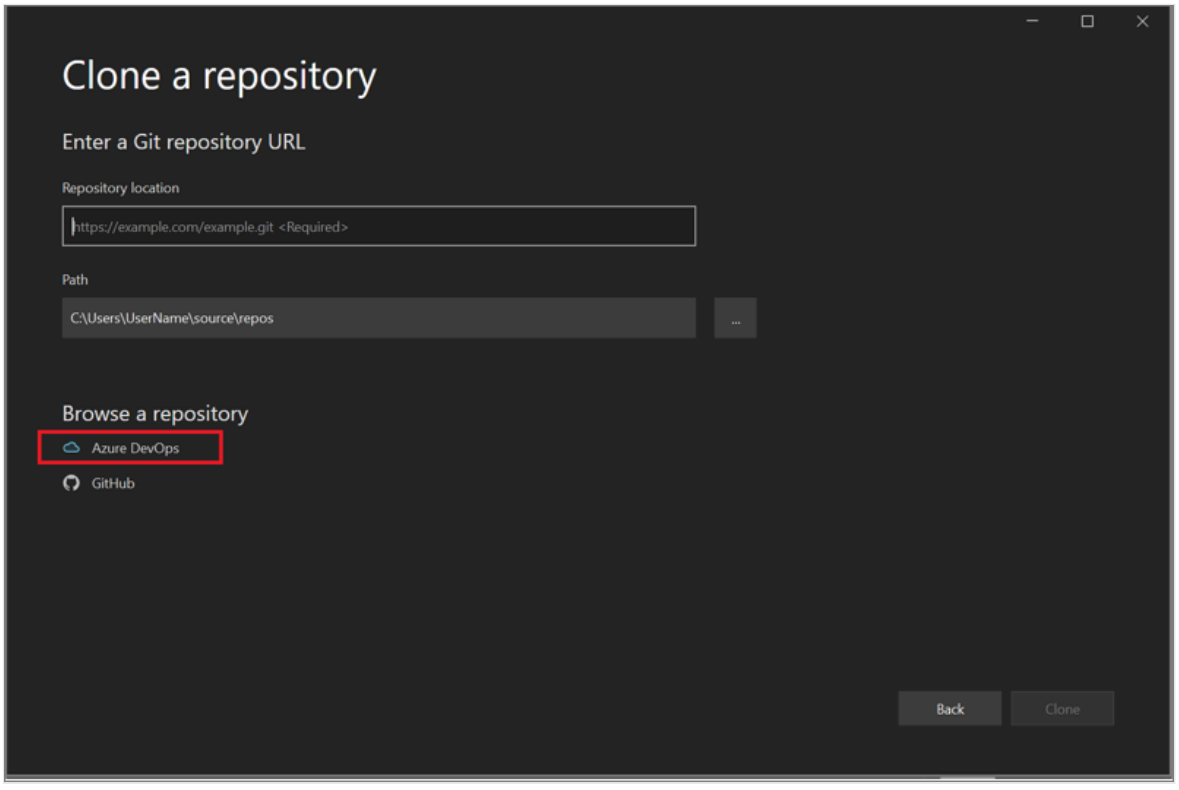
Here's how to browse to and clone an Azure DevOps repo by using Visual Studio.

1. Open Visual Studio.

2. On the start window, select **Clone a repository**.



3. In the **Browse a repository** section, select **Azure DevOps**.



4. Follow the prompts to clone an Azure DevOps repo that includes the files that you're looking for, and then open your project.

Related content

Feel free to dive into any of the following language-specific tutorials:

- Visual Studio tutorials | **C#**
- Visual Studio tutorials | **Visual Basic**
- Visual Studio tutorials | **C++**
- Visual Studio tutorials | **Python**
- Visual Studio tutorials | **JavaScript**, **TypeScript**, and **Node.js**

For more information, see:

- About Git in Visual Studio
- Brpwse a repo
- Manage a repo

Feedback

Was this page helpful? Yes No

Provide product feedback | Ask the community

Learn to use the code editor

Article • 01/24/2025

In this 10-minute introduction to the code editor in Visual Studio, we'll add code to a file to look at some of the ways that Visual Studio makes writing, navigating, and understanding code easier.

If you haven't already installed Visual Studio, go to the Visual Studio downloads page to install it for free.

This article assumes you're already familiar with a programming language. If you aren't, we suggest you look at one of the programming quickstarts first, such as create a web app with Python or C#, or create a console app with Visual Basic or C++.

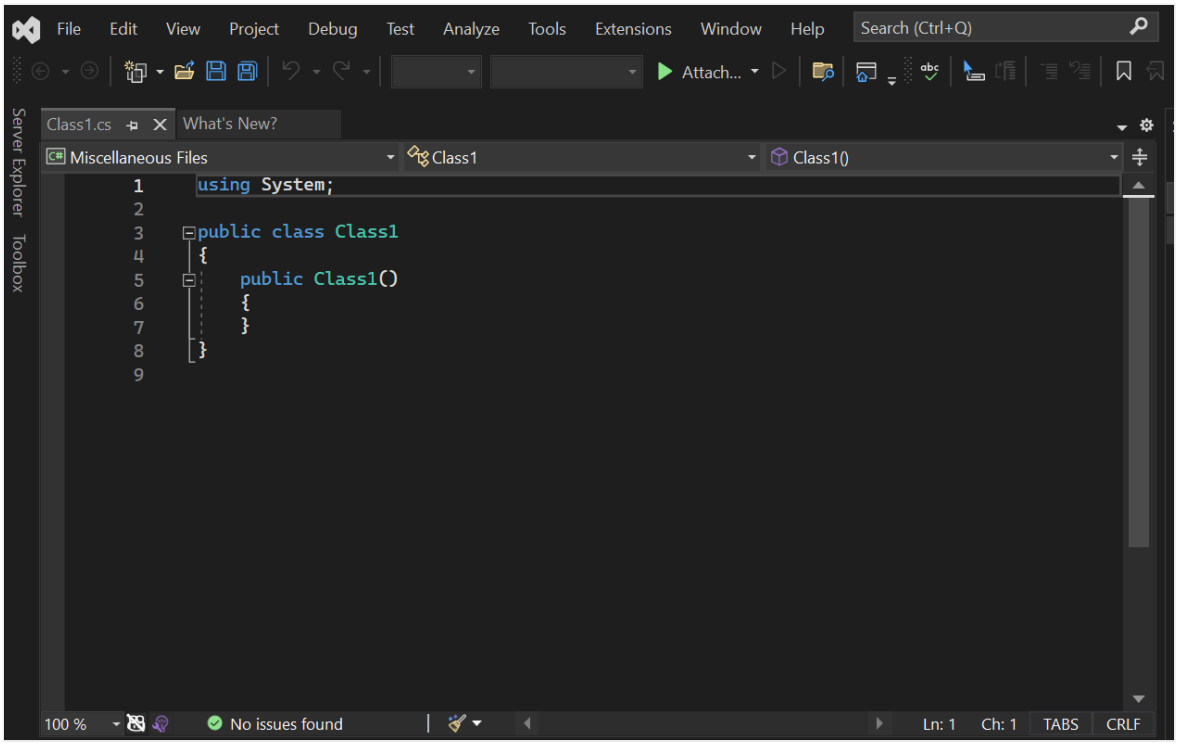
> **Tip**
>
> To follow along with this article, make sure you have the C# settings selected for Visual Studio. For information about selecting settings for the integrated development environment (IDE), see **Select environment settings**.

Create a new code file

Start by creating a new file and adding some code to it.

1. Open Visual Studio. Select the **Esc** key, or select **Continue without code** on the start window, to open the development environment.
2. From the **File** menu on the menu bar, select **New** > **File**, or select the **Ctrl**+**N** keys.
3. In the **New File** dialog box, under the **General** category, select **C# Class**, and then select **Open**.

 A new file opens in the editor with the skeleton of a C# class.



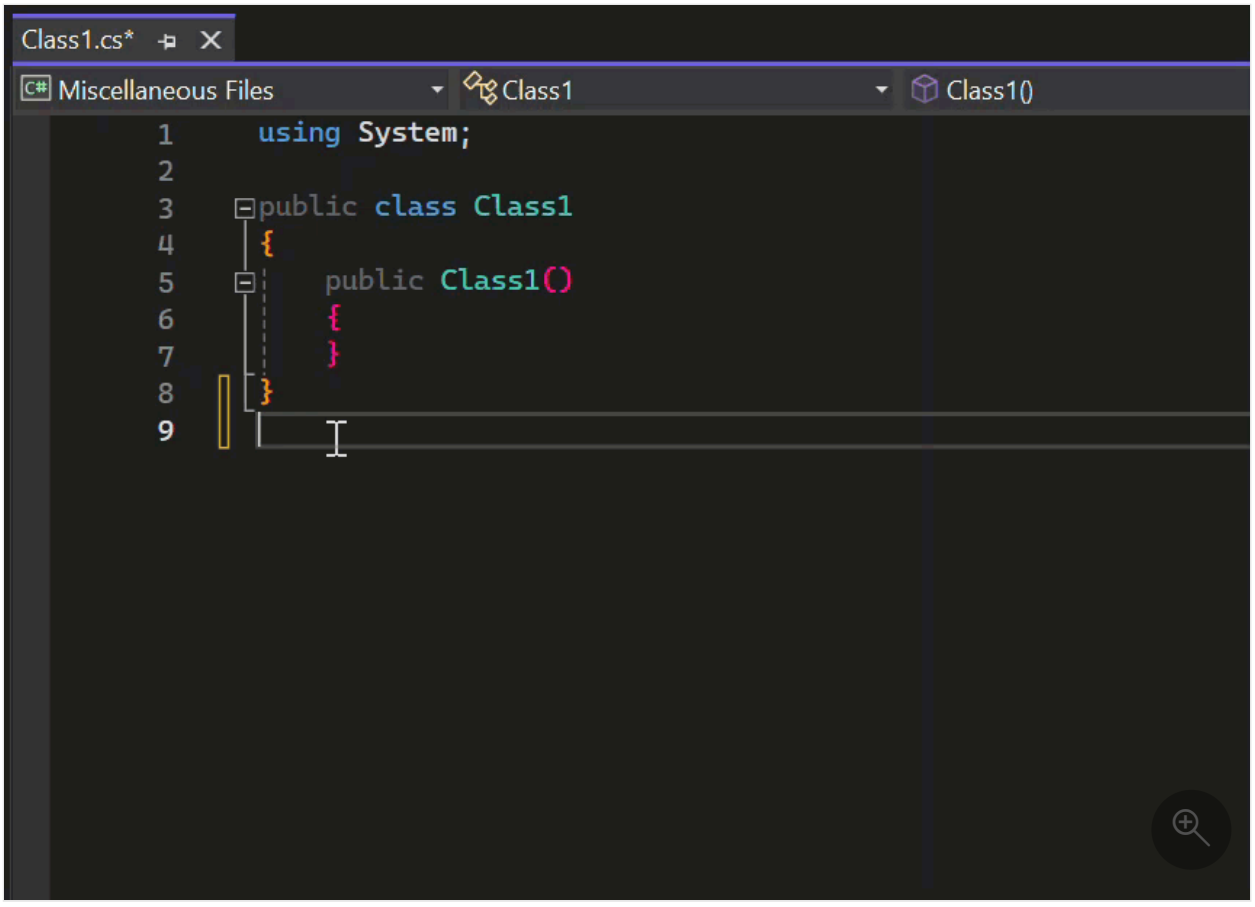
Use GitHub Copilot

GitHub Copilot acts as an AI pair programmer to provide autocomplete-style code completions and context-aware multi-line code suggestions, as you code, in real-time, right in the editor. GitHub Copilot turns natural language prompts including comments and method names into coding suggestions. You can view and incorporate suggestions from GitHub Copilot directly within the editor.

Try GitHub Copilot

Let's use Copilot to generate code suggestions:

1. Place your cursor just below the final closing brace } in the file.
2. Type a natural language comment: `// Add a method to add two numbers` and **Enter**.
3. GitHub Copilot generates a code suggestion for you. The suggested implementation shows in gray text.
4. To accept the suggestion, select **Tab**.



Let's use Copilot Chat to submit a coding-related question as a prompt:

1. Select the **GitHub Copilot** badge in the upper-right corner of the IDE.
2. Select **Open Chat Window** from the dropdown.
3. Enter the following prompt in the chat window:

 Copilot prompt

   ```
   Generate sample code for a simple C# method to add two numbers.
   ```

4. Copilot Chat generates sample code in response to your prompt.

GitHub Copilot is powered by AI, so surprises and mistakes are possible. For more information, see GitHub Copilot FAQs.

Get started with GitHub Copilot in Visual Studio. Note that it requires Visual Studio 2022 version 17.8 or later.

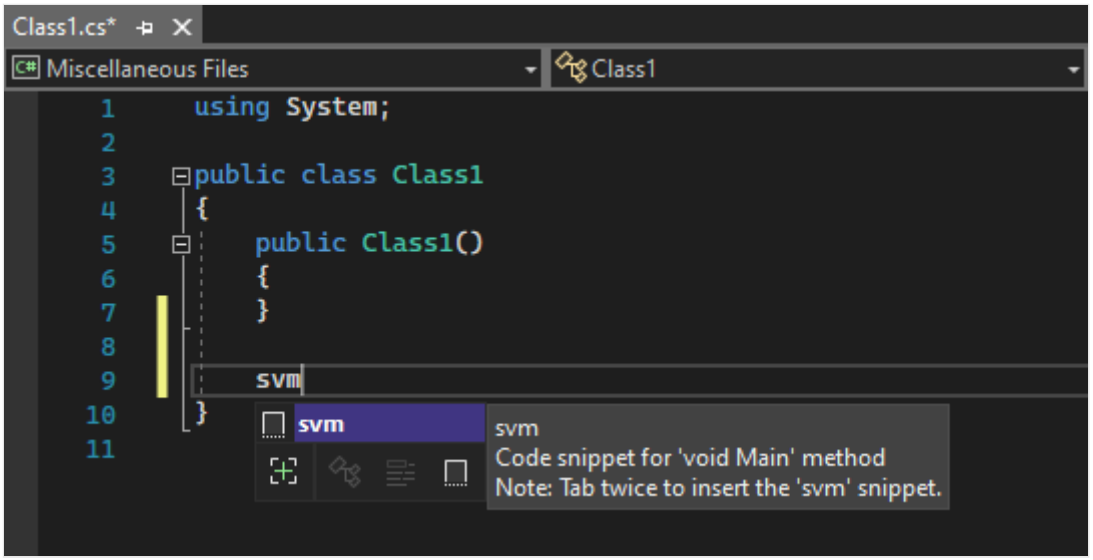
Use code snippets

Visual Studio provides useful *code snippets* that you can use to quickly and easily generate commonly used code blocks. Code snippets are available for different programming languages including C#, Visual Basic, and C++.

Let's add the C# `void Main` snippet to our file.

1. Place your cursor just above the final closing brace `}` in the file, and type the characters `svm`.

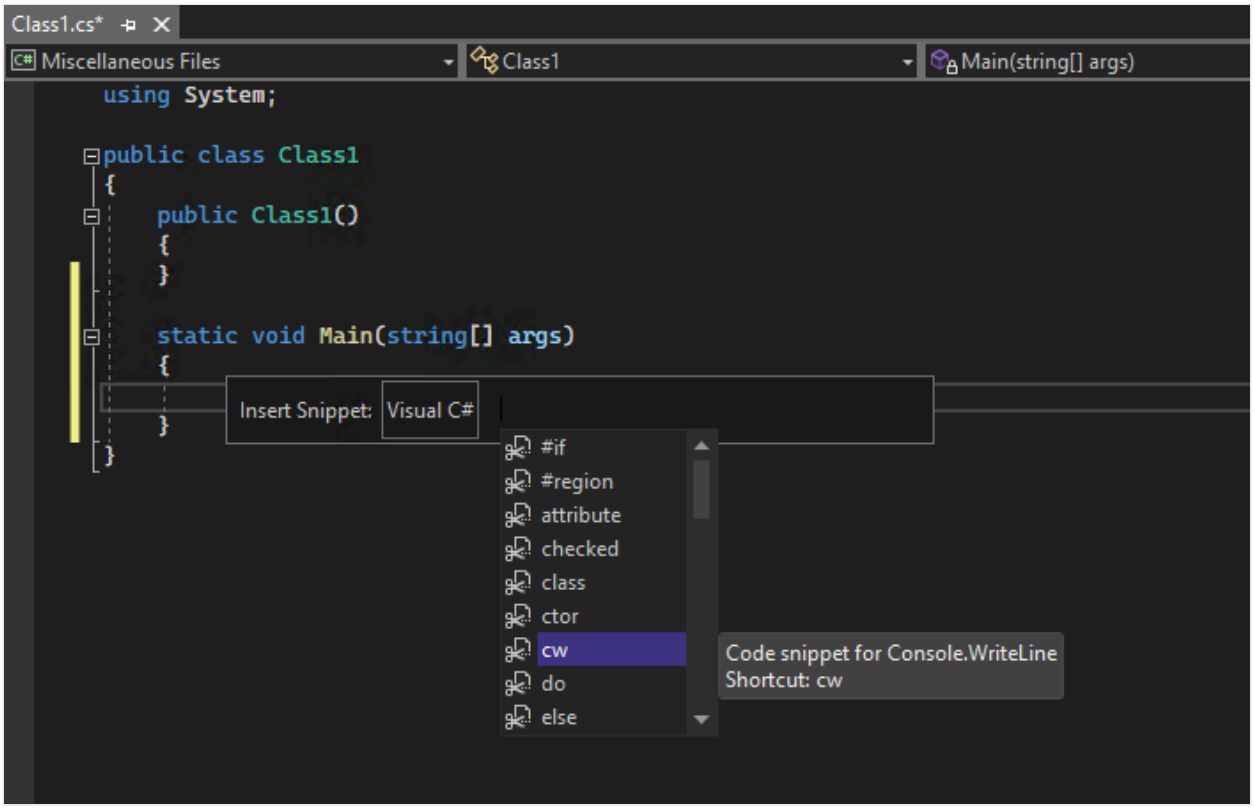
 A pop-up dialog box appears with information about the `svm` code snippet.



2. Select the **Tab** key twice to insert the code snippet.

 You'll see the `static void Main()` method signature get added to the file. The Main() method is the entry point for C# applications.

Available code snippets vary for different programming languages. You can look at the available code snippets for your language by choosing **Edit** > **IntelliSense** > **Insert Snippet** or by selecting the **Ctrl**+**K**, **Ctrl**+**X** keys, and then choosing the folder for your programming language. For C#, the snippet list looks like this:



The list includes snippets for creating a class, a constructor, a for loop, an if or switch statement, and more.

Comment out code

The Text Editor toolbar, which is the row of buttons under the menu bar in Visual Studio, helps make you more productive as you code. For example, you can toggle IntelliSense completion mode, increase or decrease a line indent, or comment out code that you don't want to compile.



Let's comment out some code.

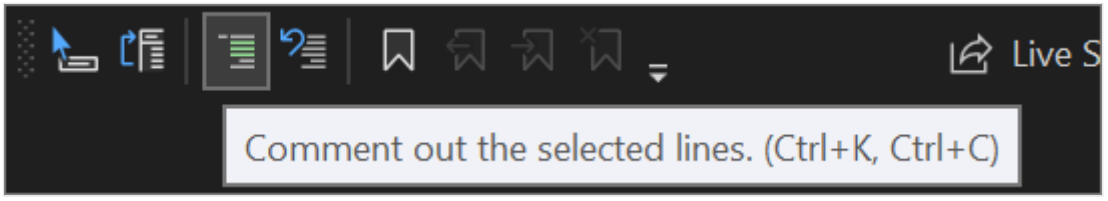
1. Paste the following code into the `Main()` method body.

```C#
// someWords is a string array.
string[] someWords = {
    "the",
    "quick",
    "brown",
    "fox",
    "jumps"
};
```

```
string[] moreWords = {
    "over",
    "the",
    "lazy",
    "dog"
};

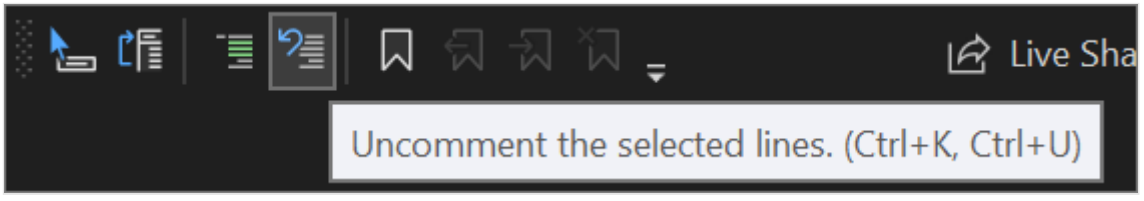
// Alphabetically sort the words.
IEnumerable<string> query = from word in someWords
                            orderby word
                            select word;
```

2. We're not using the `moreWords` variable, but we might use it later so we don't want to delete it. Instead, we'll comment out those lines. Select the entire definition of `moreWords` down to the closing semicolon, and then choose the **Comment out the selected lines** button on the Text Editor toolbar. If you prefer to use the keyboard, select **Ctrl**+**K**, **Ctrl**+**C**.



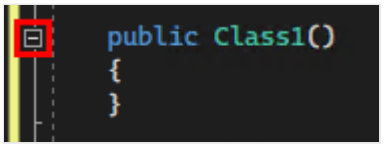
The C# comment characters `//` are added to the beginning of each selected line to comment out the code.

When you want to uncomment lines, you can select them, and then choose the **Uncomment the selected lines** button on the Text Editor toolbar. If you prefer to use the keyboard, select **Ctrl**+**K**, **Ctrl**+**U**.



Collapse code blocks

We don't want to see the empty constructor that was generated for `Class1`, so to unclutter our view of the code, let's collapse it. Choose the small gray box with the minus sign inside it in the margin of the first line of the constructor. Or, if you prefer to use the keyboard, place the cursor anywhere in the constructor code and select the **Ctrl**+**M**, **Ctrl**+**M** keys.



The code block collapses to just the first line, followed by an ellipsis (`...`). To expand the code block again, select the same gray box that now has a plus sign in it, or select **Ctrl**+**M**, **Ctrl**+**M** again. This feature is called Outlining and is especially useful when you're collapsing long methods or entire classes.

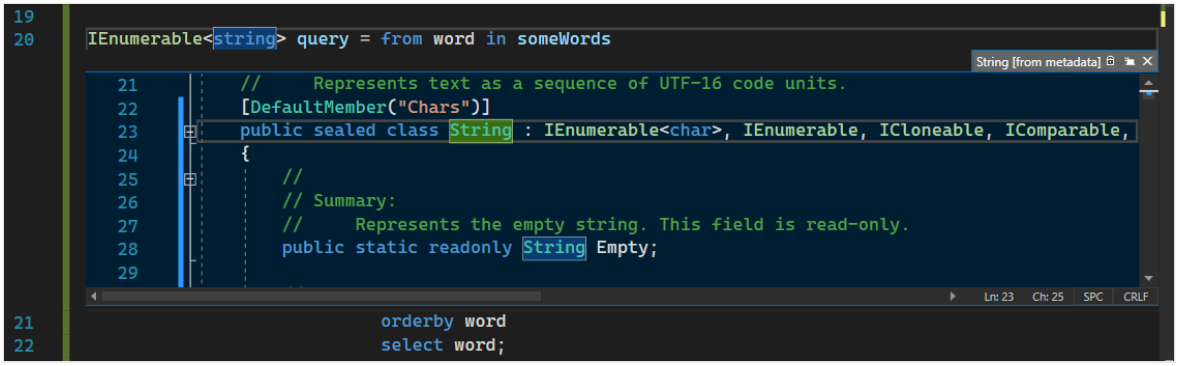
View symbol definitions

The Visual Studio editor makes it easy to inspect the definition of a type, method, or variable. One way is to go to the definition, in whichever file has it, by choosing **Go to Definition** or by selecting the **F12** key anywhere a symbol is referenced. An even quicker way that doesn't move your focus away from the code you're working on is to use Peek Definition.

Let's peek at the definition of the `string` type.

1. Right-click on any occurrence of `string` and choose **Peek Definition** from the content menu. Or, select the **Alt**+**F12** keys.

 A pop-up window appears with the definition of the `String` class. You can scroll within the pop-up window, or even peek at the definition of another type from the peeked code.



2. Close the peek definition window by choosing the small box with an "x" at the top right of the pop-up window.

Use IntelliSense to complete words

IntelliSense is an invaluable resource when you're coding. It can show you information about available members of a type, or parameter details for different overloads of a

method. You can also use IntelliSense to complete a word after you type enough characters to disambiguate it.

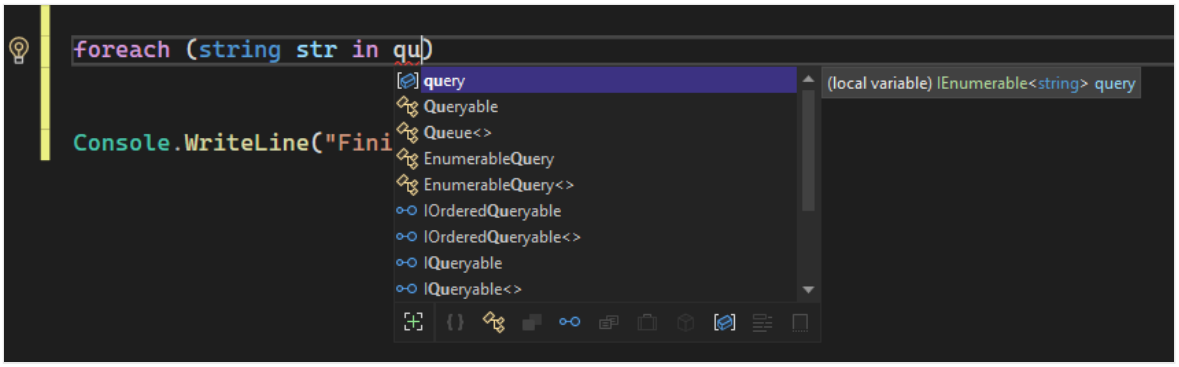
Let's add a line of code to print out the ordered strings to the console window, which is the standard place for output from the program to go.

1. Below the `query` variable, start typing the following code:

   ```
   C#

   foreach (string str in qu
   ```

 You'll see an IntelliSense pop-up appear with information about the `query` symbol.

2. To insert the rest of the word `query` by using IntelliSense word completion, select the **Tab** key.

3. Finish off the code block to look like the following code. You can practice further with code snippets by entering `cw` and then selecting **Tab** twice to generate the `Console.WriteLine` statement.

   ```
   C#

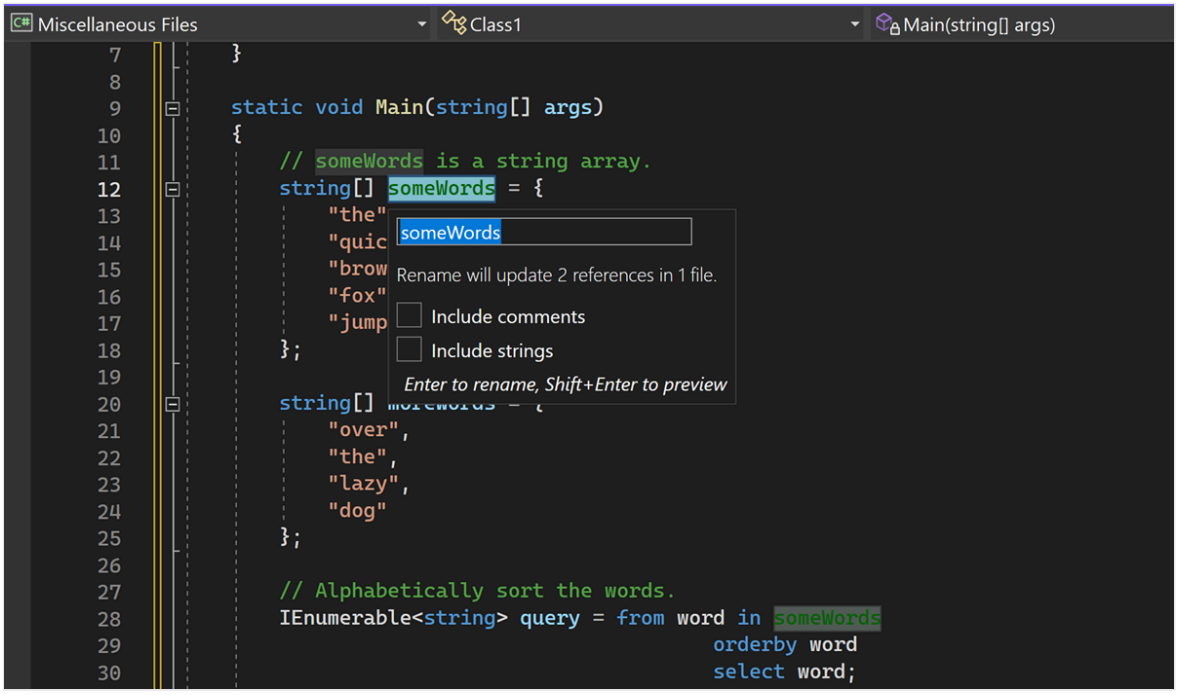
   foreach (string str in query)
   {
      Console.WriteLine(str);
   }
   ```

Refactor a name

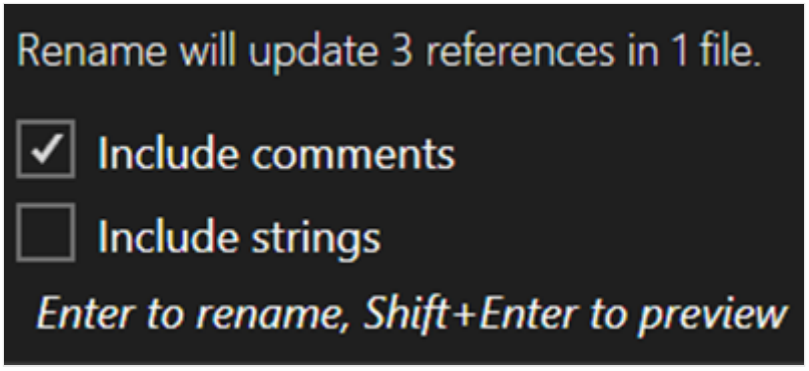
Nobody gets code right the first time, and one of the things you might have to change is the name of a variable or method. Let's try out Visual Studio's refactor functionality to rename the `someWords` variable to `unsortedWords`.

1. Place your cursor over the definition of the `someWords` variable, and choose **Rename** from the right-click or context menu, or select the **F2** key.

 A **Rename** dialog box appears at the top right of the editor.



2. Enter the desired name **unsortedWords**. You'll see that the reference to `unsortedWords` in the `query` assignment statement is also automatically renamed. Before you select the **Enter** key, select the **Include comments** checkbox in the **Rename** pop-up box.



3. Select the **Enter** key.

 Both occurrences of `someWords` in your code have been renamed, as well as the text `someWords` in your code comment.

Next steps

Learn about projects and solutions

See also

- GitHub Copilot Completions in Visual Studio
- GitHub Copilot Chat in Visual Studio
- Code snippets
- Navigate code
- Outlining
- Go To Definition and Peek Definition
- Refactoring
- Use IntelliSense

Feedback

Was this page helpful? Yes No

Provide product feedback | Ask the community

Compile and build in Visual Studio

Article • 02/03/2025

For a first introduction to building within the IDE, see Walkthrough: Building an application.

You can use any of the following methods to build an application: the Visual Studio IDE, the MSBuild command-line tools, and Azure Pipelines:

Expand table

Build Method	Benefits
IDE	- Create builds immediately and test them in a debugger. - Run multi-processor builds for C++ and C# projects. - Customize different aspects of the build system.
CMake	- Build C++ projects using the CMake tool - Use the same build system across Linux and Windows platforms.
MSBuild command line	- Build projects without installing Visual Studio. - Run multi-processor builds for all project types. - Customize most areas of the build system.
Azure Pipelines	- Automate your build process as part of a continuous integration/continuous delivery pipeline. - Apply automated tests with every build. - Employ virtually unlimited cloud-based resources for build processes. - Modify the build workflow and create build activities to perform deeply customized tasks.

The documentation in this section goes into further details of the IDE-based build process. For more information on the other methods, see CMake, MSBuild and Azure Pipelines, respectively.

Building from the IDE

When you create a project, Visual Studio created default build configurations for the project and the solution that contains the project. These configurations define how the solutions and projects are built and deployed. Project configurations in particular are unique for a target platform (such as Windows or Linux) and build type (such as debug or release). You can edit these configurations however you like, and can also create your own configurations as needed.

For a first introduction to building within the IDE, see Walkthrough: Building an application.

Next, see Building and cleaning projects and solutions in Visual Studio to learn about the different customizations you can make to the process. Customizations include changing output directories, specifying custom build events, managing project dependencies, managing build log files, and suppressing compiler warnings.

From there, you can explore a variety of other tasks:

- Understand build configurations
- Configure projects to target platforms
- Manage project and solution properties.
- Specify build events in C# and Visual Basic
- Set build options
- Build multiple projects in parallel

Related content

- Building (compiling) website projects
- CMake projects in Visual Studio

Feedback

Was this page helpful? Yes No

Provide product feedback | Ask the community

Quickstart: Debug with C++ using the Visual Studio debugger

Article • 01/12/2024

The Visual Studio debugger provides many powerful features to help you debug your apps. This topic provides a quick way to learn some of the basic features.

Create a new project

1. Open Visual Studio and create a project.

 Press **Esc** to close the start window. Type **Ctrl + Q** to open the search box, type **c++**, choose **Templates**, then choose **Create new Console App project**. In the dialog box that appears, choose **Create**.

 If you don't see the **Windows Console Application** project template, go to **Tools** > **Get Tools and Features...**, which opens the Visual Studio Installer. The Visual Studio Installer launches. Choose the **Desktop development with C++** workload, then choose **Modify**.

 Visual Studio creates the project.

2. In MyDbgApp.cpp, replace the following code

   ```
   C++

   int main()
   {
       return 0;
   }
   ```

 with this code (do not remove `#include "stdafx.h"`):

   ```
   C++

   #include <list>
   #include <iostream>

   using namespace std;

   void doWork()
   {
       list <int> c1;
   ```

```
    c1.push_back(10);
    c1.push_back(20);

    const list <int> c2 = c1;
    const int &i = c2.front();
    const int &j = c2.front();
    cout << "The first element is " << i << endl;
    cout << "The second element is " << j << endl;

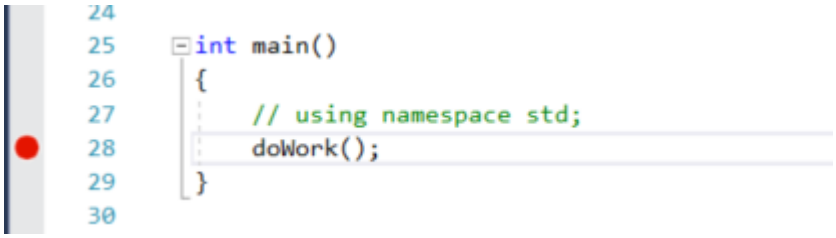
}

int main()
{
    doWork();
}
```

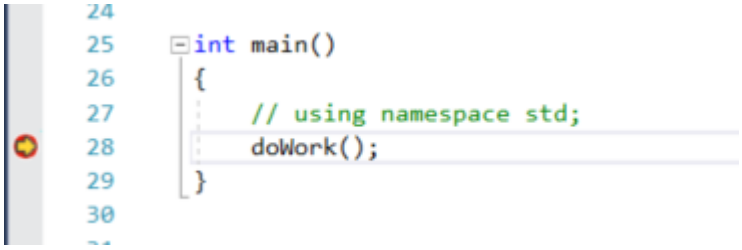
Set a breakpoint

A *breakpoint* is a marker that indicates where Visual Studio should suspend your running code so you can take a look at the values of variables, or the behavior of memory, or whether or not a branch of code is getting run. It is the most basic feature in debugging.

1. To set the breakpoint, click in the gutter to the left of the `doWork` function call (or select the line of code and press **F9**).



2. Now press **F5** (or choose **Debug** > **Start Debugging**).



 The debugger pauses where you set the breakpoint. The statement where the debugger and app execution is paused is indicated by the yellow arrow. The line with the `doWork` function call has not yet executed.

> **Tip**

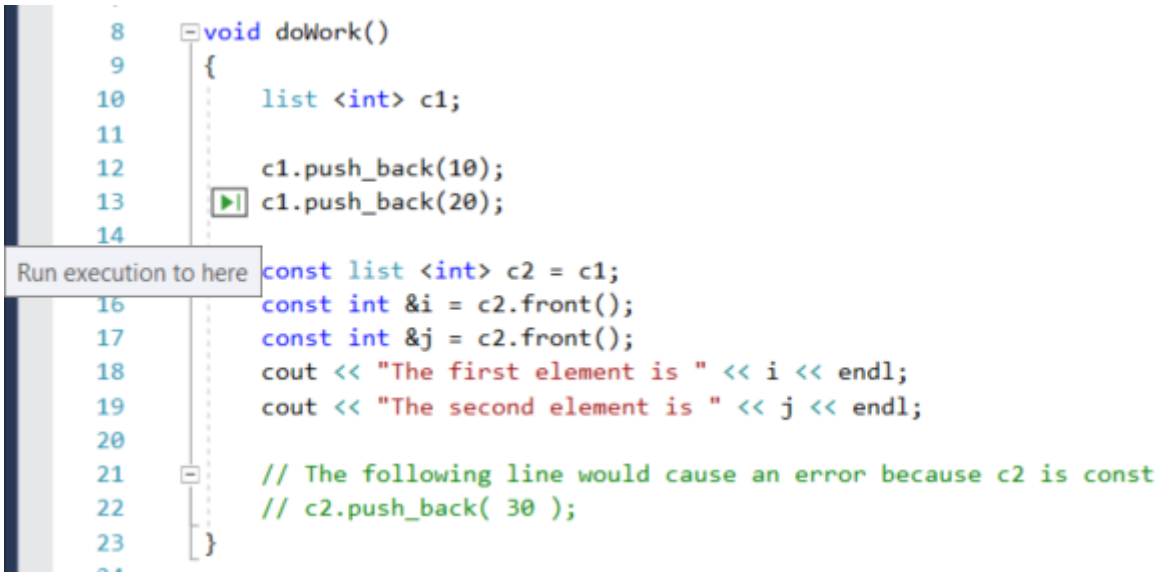
> If you have a breakpoint in a loop or recursion, or if you have many breakpoints that you frequently step through, use a **conditional breakpoint** to make sure that your code is suspended ONLY when specific conditions are met. A conditional breakpoint saves time and can also make it easier to debug issues that are hard to reproduce.

When trying to debug memory-related failures in C++, you can also use breakpoints to inspect address values (look for NULL) and reference counts.

Navigate code

There are different commands to instruct the debugger to continue. We show a useful code navigation command that is available starting in Visual Studio 2017.

While paused at the breakpoint, hover over the statement `c1.push_back(20)` until the green **Run to click** button appears, and then press the **Run to click** button.



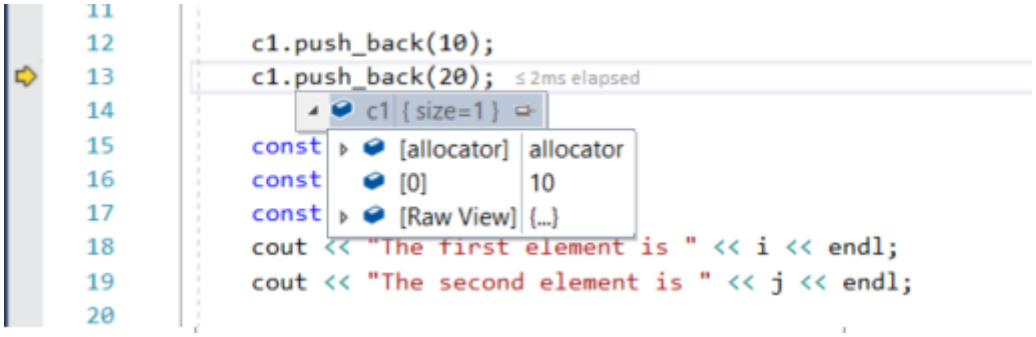

```
8   void doWork()
9   {
10      list <int> c1;
11
12      c1.push_back(10);
13      c1.push_back(20);
14
15      const list <int> c2 = c1;
16      const int &i = c2.front();
17      const int &j = c2.front();
18      cout << "The first element is " << i << endl;
19      cout << "The second element is " << j << endl;
20
21      // The following line would cause an error because c2 is const
22      // c2.push_back( 30 );
23  }
```

The app continues execution, calling `doWork`, and pauses on the line of code where you clicked the button.

Common keyboard commands used to step through code include **F10** and **F11**. For more in-depth instructions, see First look at the debugger.

Inspect variables in a datatip

1. In the current line of code (marked by the yellow execution pointer), hover over the `c1` object with your mouse to show a datatip.



The datatip shows you the current value of the `c1` variable and allows you to inspect its properties. When debugging, if you see a value you don't expect, you probably have a bug in the preceding or calling lines of code.

2. Expand the datatip to look at the current property values of the `c1` object.

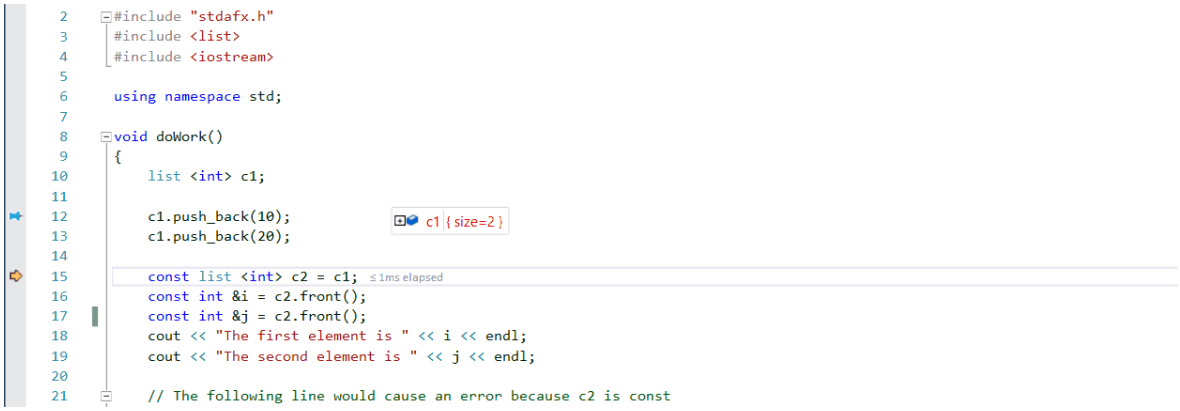
3. If you want to pin the datatip so that you can continue to see the value of `c1` while you execute code, click the small pin icon. (You can move the pinned datatip to a convenient location.)

Edit code and continue debugging

If you identify a change that you want to test in your code while in the middle of a debugging session, you can do that, too.

1. Click the second instance of `c2.front()` and change `c2.front()` to `c2.back()`.

2. Press **F10** (or **Debug** > **Step Over**) a few times to advance the debugger and execute the edited code.



F10 advances the debugger one statement at a time, but steps over functions instead of stepping into them (the code that you skip still executes).

For more information on using edit-and-continue and on feature limitations, see Edit and Continue.

Next steps

In this tutorial, you've learned how to start the debugger, step through code, and inspect variables. You may want to get a high-level look at debugger features along with links to more information.

First look at the debugger

Feedback

Was this page helpful? Yes No

Write unit tests for C/C++ in Visual Studio

Article • 12/16/2024

You can write and run your C++ unit tests by using the **Test Explorer** window. It works just like it does for other languages. For more information about using **Test Explorer**, see Run unit tests with Test Explorer.

> **Note**
>
> Some features such as Live Unit Testing, Coded UI Tests and IntelliTest aren't supported for C++.

Visual Studio includes these C++ test frameworks with no extra downloads required:

- Microsoft Unit Testing Framework for C++
- Google Test
- Boost.Test
- CTest

You can use the installed frameworks, or write your own test adapter for whatever framework you want to use within Visual Studio. A test adapter integrates unit tests with the **Test Explorer** window. Several non-Microsoft adapters are available on the Visual Studio Marketplace. For more information, see Install unit test frameworks.

- Visual Studio 2017 and later (Professional and Enterprise)

 C++ unit test projects support CodeLens.

- Visual Studio 2017 and later (all editions)

 - **Google Test Adapter** is included as a default component of the **Desktop development with C++** workload. It has a project template that you can add to a solution. Right-click on the solution node in **Solution Explorer** and choose **Add** > **New Project** on the shortcut menu to add the project template. It also has options you can configure by using **Tools** > **Options**. For more information, see How to: Use Google Test in Visual Studio.

 - **Boost.Test** is included as a default component of the **Desktop development with C++** workload. It's integrated with **Test Explorer**, but currently doesn't

have a project template. You must manually configure it. For more information, see How to: Use Boost.Test in Visual Studio.

 - **CTest** support is included with the **C++ CMake tools** component, which is part of the **Desktop development with C++** workload. For more information, see How to: Use CTest in Visual Studio.

- Earlier versions of Visual Studio

 You can download the Google Test adapter and Boost.Test Adapter extensions on the Visual Studio Marketplace. Find them at Test adapter for Boost.Test and Test adapter for Google Test.

> **Tip**
>
> You can also use Copilot `/tests` slash command to generate unit tests from code. For example, you can type `/tests using Boost framework` to generate Boost.Test tests. For more information, see **Use slash commands in Copilot Chat**.

Basic test workflow

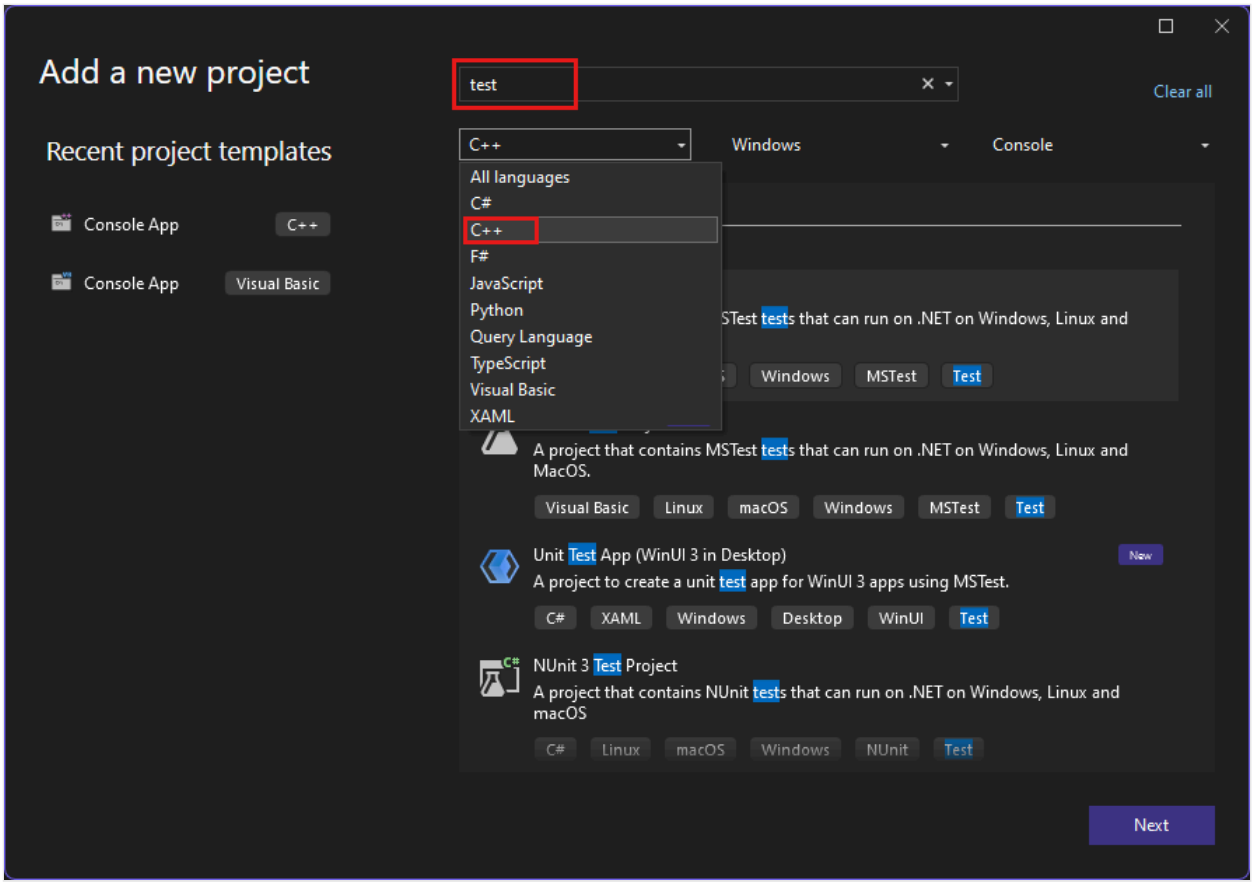
The following sections show the basic steps to get you started with C++ unit testing. The basic configuration is similar for both the Microsoft and Google Test frameworks. Boost.Test requires that you manually create a test project.

Create a test project in Visual Studio 2022

Define and run unit tests inside one or more **test projects**. A test project creates a separate app that calls the code in your executable and reports on its behavior. Create test projects in the same solution as the code you want to test.

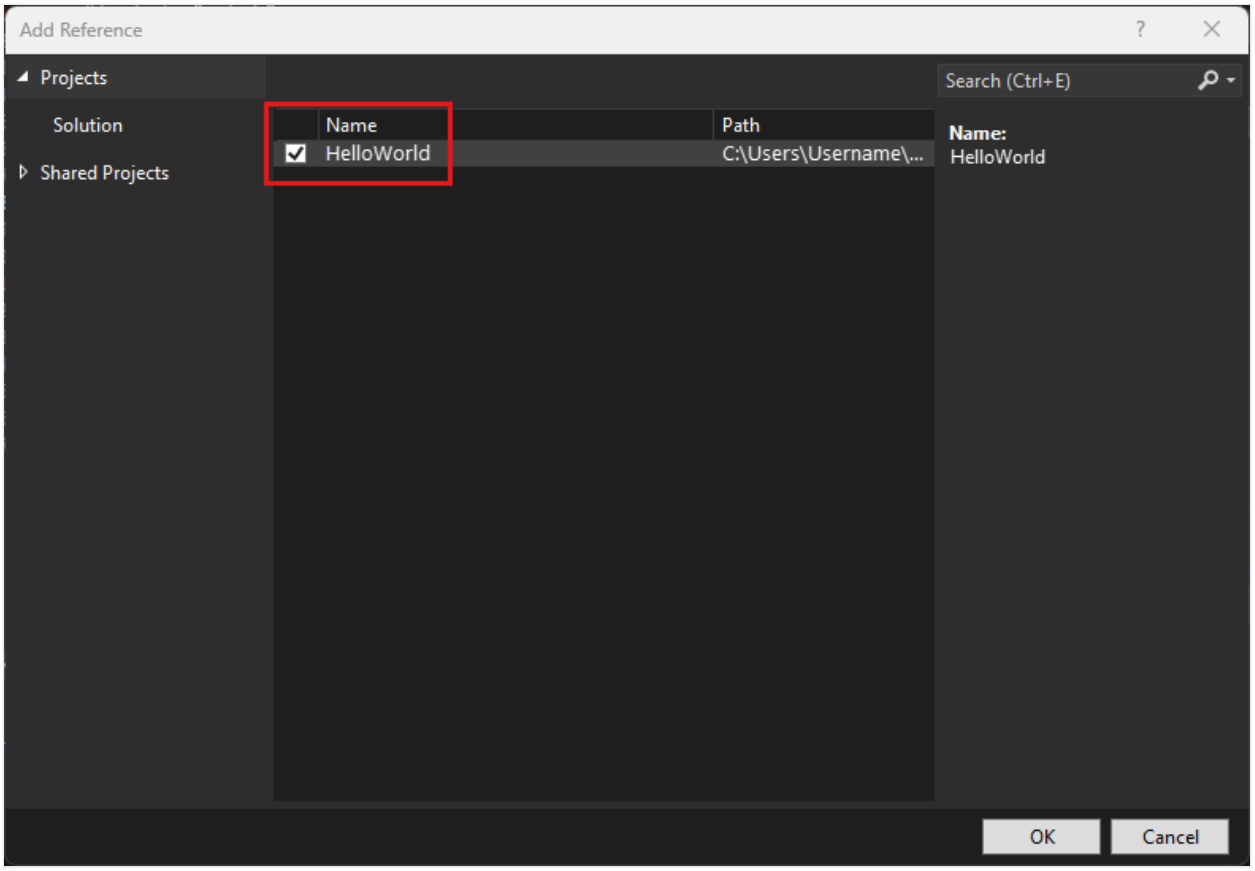
To add a new test project to an existing solution:

1. Right-click on the Solution node in **Solution Explorer**.
2. In the context menu, choose **Add** > **New Project**.
3. Set **Language** to **C++** and type *test* in the search box. The following screenshot shows the test projects that are available when the **Desktop Development with C++** and the **UWP Development** workload are installed:



Create references to other projects in the solution

To enable access to the functions in the project under test, add a reference to the project in your test project. In **Solution Explorer**, expand your test project. Right-click **References** and then select **Add** > **Reference**. In the **Add Reference** dialog box, choose the projects you want to test.

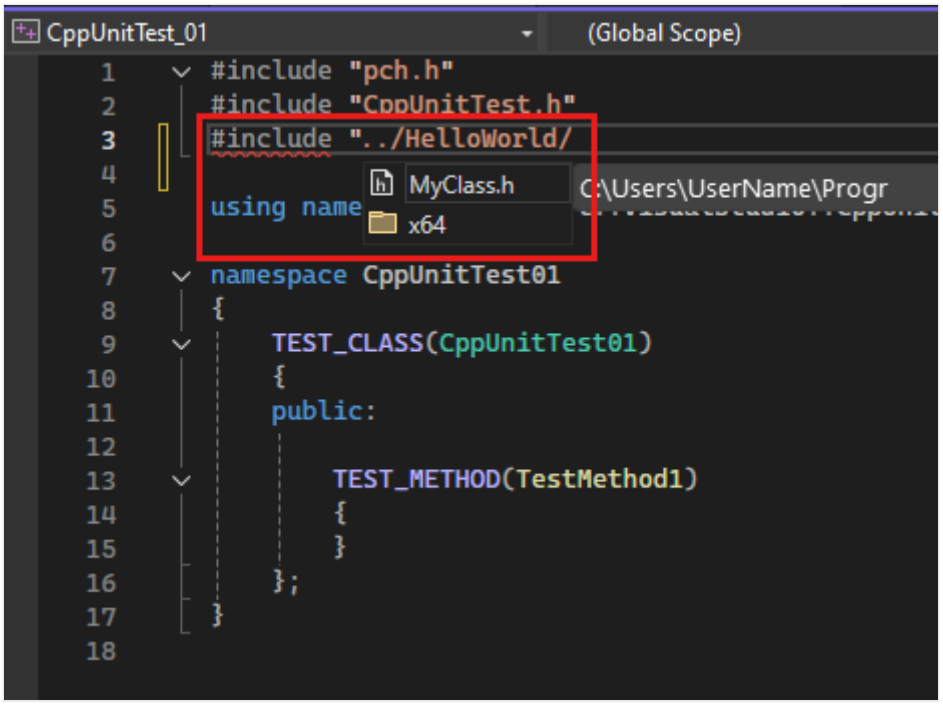


Link to object or library files

If the test code doesn't export the functions that you want to test, add the output `.obj` or `.lib` files to the dependencies of the test project. For more information, see To link the tests to the object or library files. Don't include object files that have a `main` function or another standard entry point such as `wmain`, `WinMain`, or `DllMain`. When you add new source files to your project, update the test project dependencies to include the corresponding object files.

Add #include directives for header files

In your unit test `.cpp` file, add an `#include` directive for any header files that declare the types and functions you want to test. Type `#include "`, and then IntelliSense activates to help you choose. Repeat for any more headers.

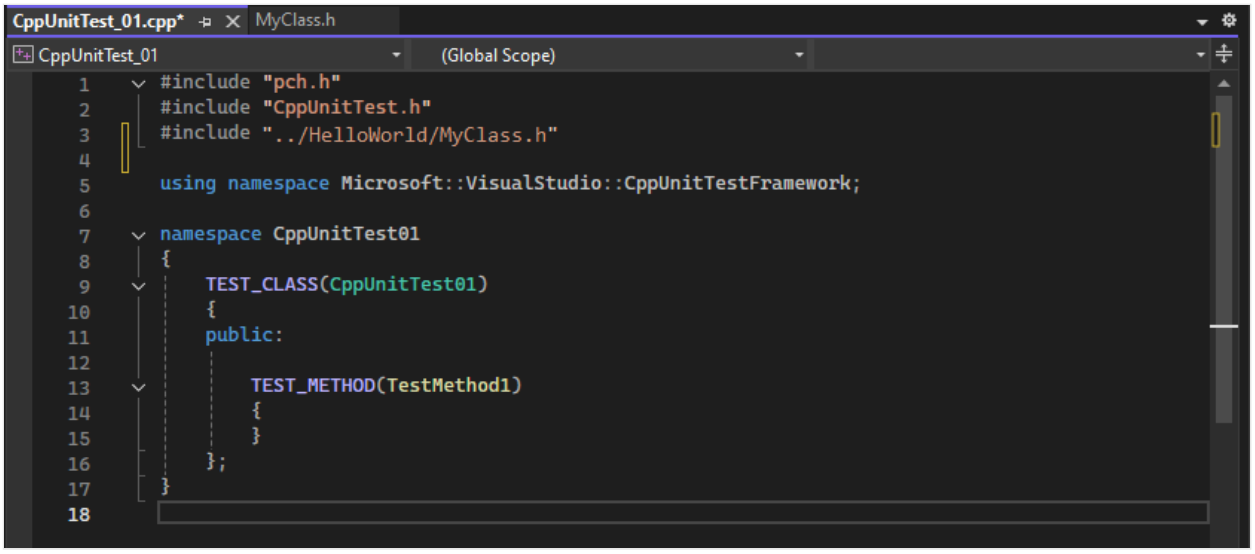


> **Tip**
>
> To avoid having to type the full path in each include statement in the source file, add the required folders in **Project** > **Properties** > **C/C++** > **General** > **Additional Include Directories**.

Write test methods

> **Note**
>
> This section shows syntax for the Microsoft Unit Testing Framework for C/C++. For more information, see Microsoft.VisualStudio.TestTools.CppUnitTestFramework API reference.
>
> For Google Test documentation, see Google Test primer. For Boost.Test, see Boost Test library: The unit test framework.

The `.cpp` file in your test project has a stub class and method defined for you. They show an example of how to write test code. The signatures use the TEST_CLASS and TEST_METHOD macros, which make the methods discoverable from the **Test Explorer** window.



TEST_CLASS and TEST_METHOD are part of the Microsoft Native Test Framework. **Test Explorer** discovers test methods in other supported frameworks in a similar way.

A TEST_METHOD returns void. To produce a test result, use the static methods in the `Assert` class to test actual results against expected results. In the following example, assume `MyClass` has a constructor that takes a `std::string`. This example shows how you can test that the constructor initializes the class the way you expect:

C++

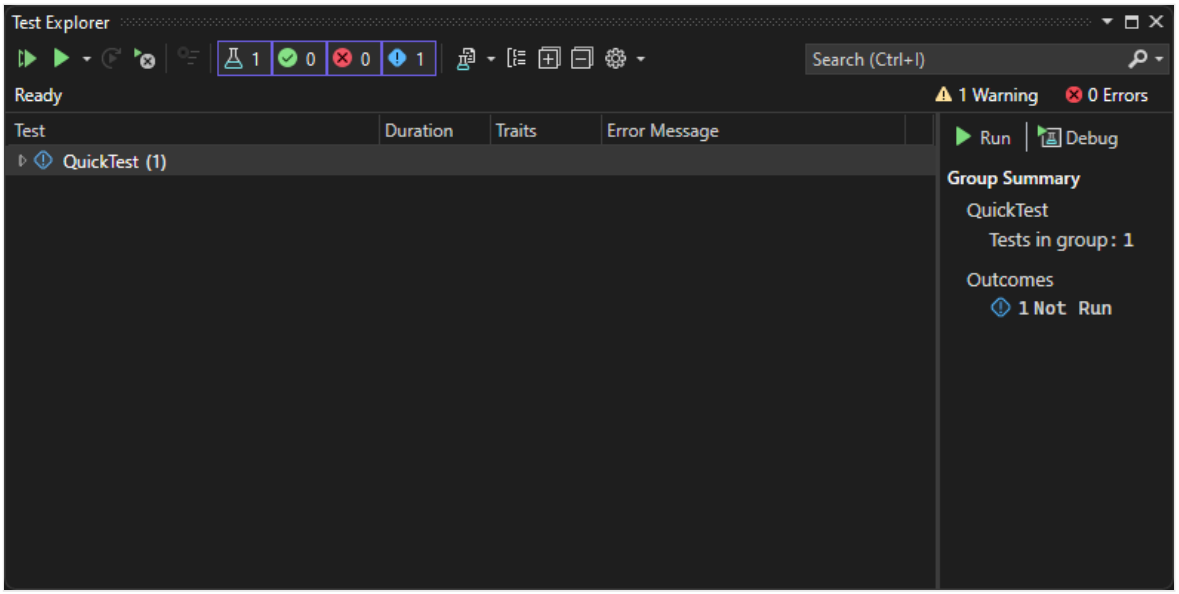
```
TEST_METHOD(TestClassInit)
{
    std::string name = "Bill";
    MyClass mc(name);
    Assert::AreEqual(name, mc.GetName());
}
```

In the previous example, the result of the `Assert::AreEqual` call determines whether the test passes or fails. The `Assert` class contains many other methods to compare expected results with actual results.

You can add *traits* to test methods to specify test owners, priority, and other information. You can then use these values to sort and group tests in **Test Explorer**. For more information, see Run unit tests with Test Explorer.

Run the tests

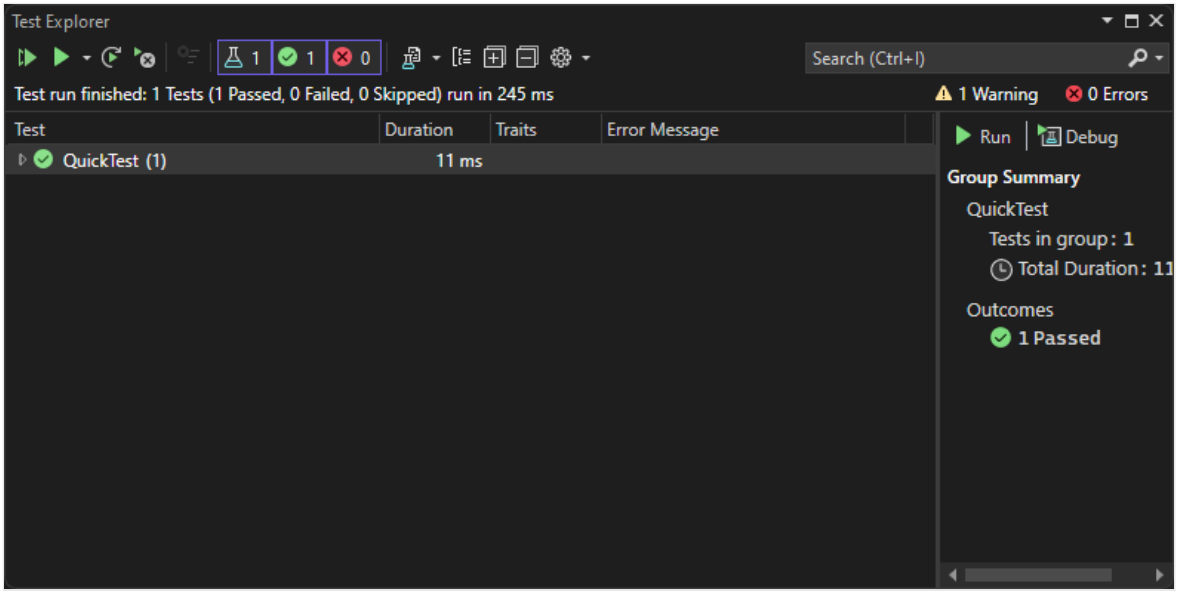
1. On the **Test** menu, choose **Test Explorer**. The following illustration shows a test project before you run tests.



> **Note**
>
> CTest integration with **Test Explorer** is not yet available. Run CTest tests from the CMake main menu.

2. If any of your tests are missing from the window, build the test project by right-clicking its node in **Solution Explorer** and choosing **Build** or **Rebuild**.

3. In **Test Explorer**, choose **Run All**, or select the specific tests you want to run. Right-click on a test for other options, including running it in debug mode with breakpoints enabled. After all the tests run, the window shows the tests that passed and the ones that failed.



For failed tests, the message displays details that help to diagnose the cause. Right-click on the failing test for a pop-up menu. Choose **Debug** to step through the function where the failure occurred.

For more information on using **Test Explorer**, see Run unit tests with Test Explorer.

For more information on unit testing, see Unit test basics.

Use CodeLens

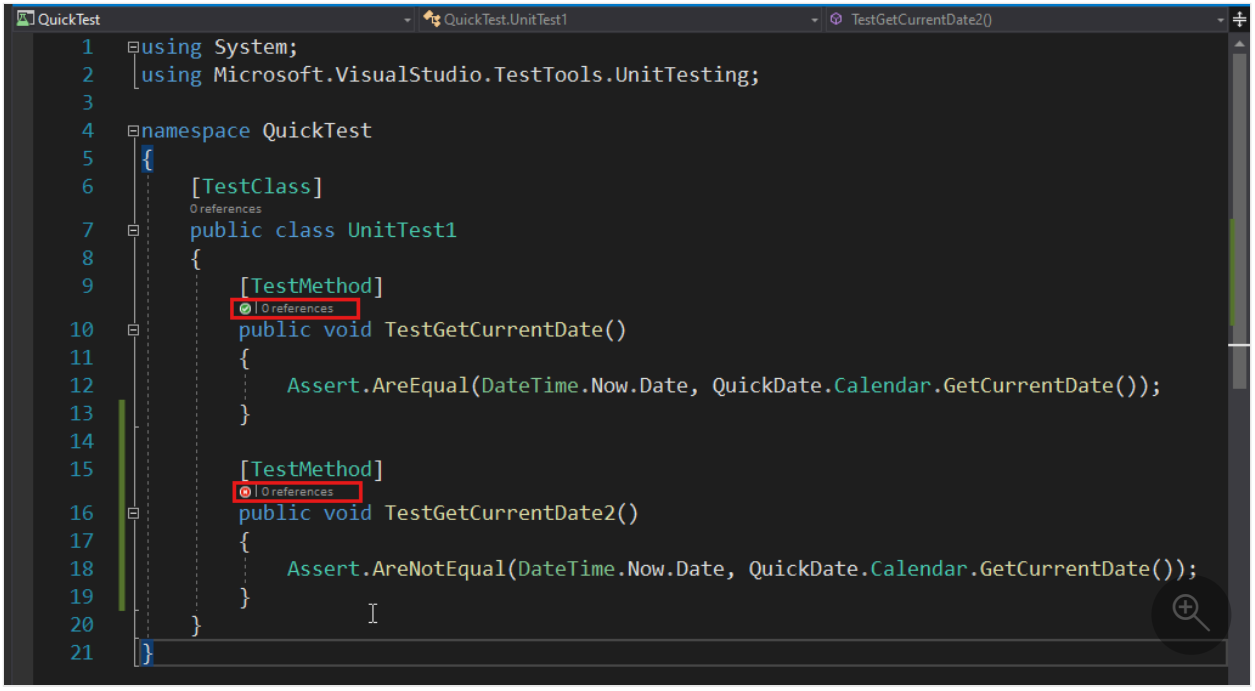
Visual Studio 2017 and later (Professional and Enterprise editions)

CodeLens lets you quickly see the status of a unit test without leaving the code editor.

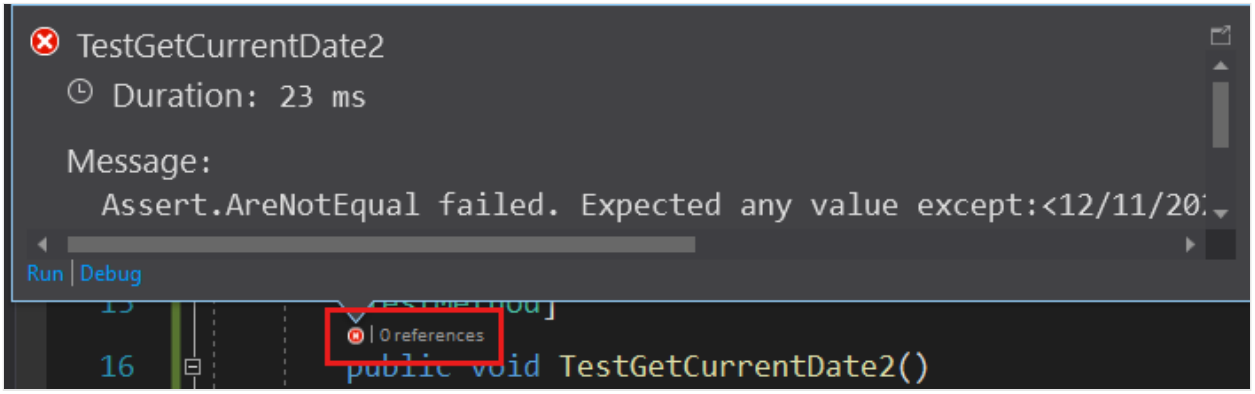
Initialize CodeLens for a C++ unit test project in any of the following ways:

- Edit and build your test project or solution.
- Rebuild your project or solution.
- Run tests from the **Test Explorer** window.

After you initialize CodeLens, you can see the test status icons above each unit test.



Choose the icon for more information, or to run or debug the unit test:



Related content

- Unit test your code

Feedback

Was this page helpful? Yes No

Provide product feedback | Ask the community

Walkthrough: Compiling a Native C++ Program on the Command Line

Article • 02/08/2022

Visual Studio includes a command-line C and C++ compiler. You can use it to create everything from basic console apps to Universal Windows Platform apps, Desktop apps, device drivers, and .NET components.

In this walkthrough, you create a basic, "Hello, World"-style C++ program by using a text editor, and then compile it on the command line. If you'd like to try the Visual Studio IDE instead of using the command line, see Walkthrough: Working with Projects and Solutions (C++) or Using the Visual Studio IDE for C++ Desktop Development.

In this walkthrough, you can use your own C++ program instead of typing the one that's shown. Or, you can use a C++ code sample from another help article.

Prerequisites

To complete this walkthrough, you must have installed either Visual Studio and the optional **Desktop development with C++** workload, or the command-line Build Tools for Visual Studio.

Visual Studio is an *integrated development environment* (IDE). It supports a full-featured editor, resource managers, debuggers, and compilers for many languages and platforms. Versions available include the free Visual Studio Community edition, and all can support C and C++ development. For information on how to download and install Visual Studio, see Install C++ support in Visual Studio.

The Build Tools for Visual Studio installs only the command-line compilers, tools, and libraries you need to build C and C++ programs. It's perfect for build labs or classroom exercises and installs relatively quickly. To install only the command-line tools, look for Build Tools for Visual Studio on the Visual Studio Downloads page.

Before you can build a C or C++ program on the command line, verify that the tools are installed, and you can access them from the command line. Visual C++ has complex requirements for the command-line environment to find the tools, headers, and libraries it uses. **You can't use Visual C++ in a plain command prompt window** without doing some preparation. Fortunately, Visual C++ installs shortcuts for you to launch a developer command prompt that has the environment set up for command line builds. Unfortunately, the names of the developer command prompt shortcuts and where

they're located are different in almost every version of Visual C++ and on different versions of Windows. Your first walkthrough task is finding the right one to use.

> ⓘ **Note**
>
> A developer command prompt shortcut automatically sets the correct paths for the compiler and tools, and for any required headers and libraries. You must set these environment values yourself if you use a regular **Command Prompt** window. For more information, see **Use the MSVC toolset from the command line**. We recommend you use a developer command prompt shortcut instead of building your own.

Open a developer command prompt

1. If you have installed Visual Studio 2017 or later on Windows 10 or later, open the Start menu and choose **All apps**. Scroll down and open the **Visual Studio** folder (not the Visual Studio application). Choose **Developer Command Prompt for VS** to open the command prompt window.

 If you have installed Microsoft Visual C++ Build Tools 2015 on Windows 10 or later, open the **Start** menu and choose **All apps**. Scroll down and open the **Visual C++ Build Tools** folder. Choose **Visual C++ 2015 x86 Native Tools Command Prompt** to open the command prompt window.

 You can also use the Windows search function to search for "developer command prompt" and choose one that matches your installed version of Visual Studio. Use the shortcut to open the command prompt window.

2. Next, verify that the Visual C++ developer command prompt is set up correctly. In the command prompt window, enter `cl` and verify that the output looks something like this:

 Output

   ```
   C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise>cl
   Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
   Copyright (C) Microsoft Corporation.  All rights reserved.

   usage: cl [ option... ] filename... [ /link linkoption... ]
   ```

 There may be differences in the current directory or version numbers. These values depend on the version of Visual C++ and any updates installed. If the above

output is similar to what you see, then you're ready to build C or C++ programs at the command line.

> **Note**
>
> If you get an error such as "'cl' is not recognized as an internal or external command, operable program or batch file," error C1034, or error LNK1104 when you run the `cl` command, then either you are not using a developer command prompt, or something is wrong with your installation of Visual C++. You must fix this issue before you can continue.

If you can't find the developer command prompt shortcut, or if you get an error message when you enter `cl`, then your Visual C++ installation may have a problem. Try reinstalling the Visual C++ component in Visual Studio, or reinstall the Microsoft Visual C++ Build Tools. Don't go on to the next section until the `cl` command works. For more information about installing and troubleshooting Visual C++, see Install Visual Studio.

> **Note**
>
> Depending on the version of Windows on the computer and the system security configuration, you might have to right-click to open the shortcut menu for the developer command prompt shortcut and then choose **Run as administrator** to successfully build and run the program that you create by following this walkthrough.

Create a Visual C++ source file and compile it on the command line

1. In the developer command prompt window, enter `md c:\hello` to create a directory, and then enter `cd c:\hello` to change to that directory. This directory is where both your source file and the compiled program get created.

2. Enter `notepad hello.cpp` in the command prompt window.

 Choose **Yes** when Notepad prompts you to create a new file. This step opens a blank Notepad window, ready for you to enter your code in a file named hello.cpp.

3. In Notepad, enter the following lines of code:

   ```
   ```

C++

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello, world, from Visual C++!" << endl;
}
```

This code is a simple program that will write one line of text on the screen and then exit. To minimize errors, copy this code and paste it into Notepad.

4. Save your work! In Notepad, on the **File** menu, choose **Save**.

 Congratulations, you've created a C++ source file, hello.cpp, that is ready to compile.

5. Switch back to the developer command prompt window. Enter `dir` at the command prompt to list the contents of the c:\hello directory. You should see the source file hello.cpp in the directory listing, which looks something like:

Output

```
c:\hello>dir
 Volume in drive C has no label.
 Volume Serial Number is CC62-6545

 Directory of c:\hello

05/24/2016  05:36 PM    <DIR>          .
05/24/2016  05:36 PM    <DIR>          ..
05/24/2016  05:37 PM               115 hello.cpp
               1 File(s)            115 bytes
               2 Dir(s)  571,343,446,016 bytes free
```

The dates and other details will differ on your computer.

> **Note**
>
> If you don't see your source code file, *`hello.cpp`*, make sure the current working directory in your command prompt is the *`C:\hello`* directory you created. Also make sure that this is the directory where you saved your source file. And make sure that you saved the source code with a *`.cpp`* file name extension, not a *`.txt`* extension. Your source file gets saved in the current directory as a *`.cpp`* file automatically if you open Notepad at the command

prompt by using the `notepad hello.cpp` command. Notepad's behavior is different if you open it another way: By default, Notepad appends a `.txt` extension to new files when you save them. It also defaults to saving files in your *Documents* directory. To save your file with a `.cpp` extension in Notepad, choose **File** > **Save As**. In the **Save As** dialog, navigate to your `C:\hello` folder in the directory tree view control. Then use the **Save as type** dropdown control to select **All Files (*.*)**. Enter `hello.cpp` in the **File name** edit control, and then choose **Save** to save the file.

6. At the developer command prompt, enter `cl /EHsc hello.cpp` to compile your program.

 The cl.exe compiler generates an .obj file that contains the compiled code, and then runs the linker to create an executable program named hello.exe. This name appears in the lines of output information that the compiler displays. The output of the compiler should look something like:

 Output

   ```
   c:\hello>cl /EHsc hello.cpp
   Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
   Copyright (C) Microsoft Corporation.  All rights reserved.

   hello.cpp
   Microsoft (R) Incremental Linker Version 14.10.25017.0
   Copyright (C) Microsoft Corporation.  All rights reserved.

   /out:hello.exe
   hello.obj
   ```

 Note

 If you get an error such as "'cl' is not recognized as an internal or external command, operable program or batch file," error C1034, or error LNK1104, your developer command prompt is not set up correctly. For information on how to fix this issue, go back to the **Open a developer command prompt** section.

 Note

 If you get a different compiler or linker error or warning, review your source code to correct any errors, then save it and run the compiler again. For

information about specific errors, use the search box to look for the error number.

7. To run the hello.exe program, at the command prompt, enter `hello`.

 The program displays this text and exits:

   ```
   Output

   Hello, world, from Visual C++!
   ```

 Congratulations, you've compiled and run a C++ program by using the command-line tools.

Next steps

This "Hello, World" example is about as simple as a C++ program can get. Real world programs usually have header files, more source files, and link to libraries.

You can use the steps in this walkthrough to build your own C++ code instead of typing the sample code shown. These steps also let you build many C++ code sample programs that you find elsewhere. You can put your source code and build your apps in any writeable directory. By default, the Visual Studio IDE creates projects in your user folder, in a *source\repos* subfolder. Older versions may put projects in a *Documents\Visual Studio <version>\Projects* folder.

To compile a program that has additional source code files, enter them all on the command line, like:

```
cl /EHsc file1.cpp file2.cpp file3.cpp
```

The `/EHsc` command-line option instructs the compiler to enable standard C++ exception handling behavior. Without it, thrown exceptions can result in undestroyed objects and resource leaks. For more information, see /EH (Exception Handling Model).

When you supply additional source files, the compiler uses the first input file to create the program name. In this case, it outputs a program called file1.exe. To change the name to program1.exe, add an /out linker option:

```
cl /EHsc file1.cpp file2.cpp file3.cpp /link /out:program1.exe
```

And to catch more programming mistakes automatically, we recommend you compile by using either the /W3 or /W4 warning level option:

```
cl /W4 /EHsc file1.cpp file2.cpp file3.cpp /link /out:program1.exe
```

The compiler, cl.exe, has many more options. You can apply them to build, optimize, debug, and analyze your code. For a quick list, enter `cl /?` at the developer command prompt. You can also compile and link separately and apply linker options in more complex build scenarios. For more information on compiler and linker options and usage, see C/C++ Building Reference.

You can use NMAKE and makefiles, MSBuild and project files, or CMake, to configure and build more complex projects on the command line. For more information on using these tools, see NMAKE Reference, MSBuild, and CMake projects in Visual Studio.

The C and C++ languages are similar, but not the same. The MSVC compiler uses a simple rule to determine which language to use when it compiles your code. By default, the MSVC compiler treats files that end in `.c` as C source code, and files that end in `.cpp` as C++ source code. To force the compiler to treat all files as C++ independent of file name extension, use the /TP compiler option.

The MSVC compiler includes a C Runtime Library (CRT) that conforms to the ISO C99 standard, with minor exceptions. Portable code generally compiles and runs as expected. Certain obsolete library functions, and several POSIX function names, are deprecated by the MSVC compiler. The functions are supported, but the preferred names have changed. For more information, see Security Features in the CRT and Compiler Warning (level 3) C4996.

See also

C++ Language Reference
Projects and build systems
MSVC Compiler Options

Walkthrough: Compile a C program on the command line

Article • 05/10/2022

The Visual Studio build tools include a C compiler that you can use to create everything from basic console programs to full Windows Desktop applications, mobile apps, and more. Microsoft C/C++ (MSVC) is a C and C++ compiler that, in its latest versions, conforms to some of the latest C language standards, including C11 and C17.

This walkthrough shows how to create a basic, "Hello, World"-style C program by using a text editor, and then compile it on the command line. If you'd rather work in C++ on the command line, see Walkthrough: Compiling a Native C++ Program on the Command Line. If you'd like to try the Visual Studio IDE instead of using the command line, see Walkthrough: Working with Projects and Solutions (C++) or Using the Visual Studio IDE for C++ Desktop Development.

Prerequisites

To complete this walkthrough, you must have installed either Visual Studio or the Build Tools for Visual Studio and the optional Desktop development with C++ workload.

Visual Studio is a powerful integrated development environment that supports a full-featured editor, resource managers, debuggers, and compilers for many languages and platforms. For information on these features and how to download and install Visual Studio, including the free Visual Studio Community edition, see Install Visual Studio.

The Build Tools for Visual Studio version of Visual Studio installs only the command-line toolset, the compilers, tools, and libraries you need to build C and C++ programs. It's perfect for build labs or classroom exercises and installs relatively quickly. To install only the command-line toolset, download Build Tools for Visual Studio from the Visual Studio downloads page and run the installer. In the Visual Studio installer, select the **Desktop development with C++** workload (in older versions of Visual Studio, select the **C++ build tools** workload), and choose **Install**.

When you've installed the tools, there's another tool you'll use to build a C or C++ program on the command line. MSVC has complex requirements for the command-line environment to find the tools, headers, and libraries it uses. **You can't use MSVC in a plain command prompt window** without some preparation. You need a *developer command prompt* window, which is a regular command prompt window that has all the required environment variables set. Fortunately, Visual Studio installs shortcuts for you

to launch developer command prompts that have the environment set up for command line builds. Unfortunately, the names of the developer command prompt shortcuts and where they're located are different in almost every version of Visual Studio and on different versions of Windows. Your first walkthrough task is to find the right shortcut to use.

> **Note**
>
> A developer command prompt shortcut automatically sets the correct paths for the compiler and tools, and for any required headers and libraries. Some of these values are different for each build configuration. You must set these environment values yourself if you don't use one of the shortcuts. For more information, see **Use the MSVC toolset from the command line**. Because the build environment is complex, we strongly recommend you use a developer command prompt shortcut instead of building your own.

These instructions vary depending on which version of Visual Studio you're using. To see the documentation for your preferred version of Visual Studio, use the **Version** selector control. It's found at the top of the table of contents on this page.

Open a developer command prompt in Visual Studio 2022

If you've installed Visual Studio 2022 on Windows 10 or later, open the Start menu, and choose **All apps**. Then, scroll down and open the **Visual Studio 2022** folder (not the Visual Studio 2022 app). Choose **Developer Command Prompt for VS 2022** to open the command prompt window.

If you're using a different version of Windows, look in your Start menu or Start page for a Visual Studio tools folder that contains a developer command prompt shortcut. You can also use the Windows search function to search for "developer command prompt" and choose one that matches your installed version of Visual Studio. Use the shortcut to open the command prompt window.

Next, verify that the developer command prompt is set up correctly. In the command prompt window, enter `cl` (or `CL`, case doesn't matter for the compiler name, but it does matter for compiler options). The output should look something like this:

```
Output
```

```
C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise>cl
Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]
```

There may be differences in the current directory or version numbers, depending on the version of Visual Studio and any updates installed. If the above output is similar to what you see, then you're ready to build C or C++ programs at the command line.

> **Note**
>
> If you get an error such as "'cl' is not recognized as an internal or external command, operable program or batch file," error C1034, or error LNK1104 when you run the **cl** command, then either you are not using a developer command prompt, or something is wrong with your installation of Visual Studio. You must fix this issue before you can continue.

If you can't find the developer command prompt shortcut, or if you get an error message when you enter `cl`, then your Visual Studio installation may have a problem. If you're using Visual Studio 2017 or later, try reinstalling the **Desktop development with C++** workload in the Visual Studio installer. For details, see Install C++ support in Visual Studio. Or, reinstall the Build Tools from the Visual Studio downloads page. Don't go on to the next section until the `cl` command works. For more information about installing and troubleshooting Visual Studio, see Install Visual Studio.

> **Note**
>
> Depending on the version of Windows on the computer and the system security configuration, you might have to right-click to open the shortcut menu for the developer command prompt shortcut and then choose **Run as Administrator** to successfully build and run the program that you create by following this walkthrough.

Create a C source file and compile it on the command line

1. In the developer command prompt window, enter `cd c:\` to change the current working directory to the root of your C: drive. Next, enter `md c:\hello` to create a

directory, and then enter `cd c:\hello` to change to that directory. This directory will hold your source file and the compiled program.

2. Enter `notepad hello.c` at the developer command prompt. In the Notepad alert dialog that pops up, choose **Yes** to create a new *`hello.c`* file in your working directory.

3. In Notepad, enter the following lines of code:

C

```c
#include <stdio.h>

int main()
{
    printf("Hello, World! This is a native C program compiled on the
command line.\n");
    return 0;
}
```

4. On the Notepad menu bar, choose **File** > **Save** to save *`hello.c`* in your working directory.

5. Switch back to the developer command prompt window. Enter `dir` at the command prompt to list the contents of the *`c:\hello`* directory. You should see the source file *`hello.c`* in the directory listing, which looks something like:

Output

```
C:\hello>dir
 Volume in drive C has no label.
 Volume Serial Number is CC62-6545

 Directory of C:\hello

10/02/2017  03:46 PM    <DIR>          .
10/02/2017  03:46 PM    <DIR>          ..
10/02/2017  03:36 PM               143 hello.c
               1 File(s)            143 bytes
               2 Dir(s)  514,900,566,016 bytes free
```

The dates and other details will differ on your computer. If you don't see your source code file, *`hello.c`*, make sure you've changed to the *`c:\hello`* directory you created, and in Notepad, make sure that you saved your source file in this

directory. Also make sure that you saved the source code with a `.c` file name extension, not a `.txt` extension.

6. To compile your program, enter `cl hello.c` at the developer command prompt.

 You can see the executable program name, hello.exe, in the lines of output information that the compiler displays:

 Output

   ```
   c:\hello>cl hello.c
   Microsoft (R) C/C++ Optimizing Compiler Version 19.10.25017 for x86
   Copyright (C) Microsoft Corporation.  All rights reserved.

   hello.c
   Microsoft (R) Incremental Linker Version 14.10.25017.0
   Copyright (C) Microsoft Corporation.  All rights reserved.

   /out:hello.exe
   hello.obj
   ```

 > **Note**
 >
 > If you get an error such as "'cl' is not recognized as an internal or external command, operable program or batch file," error C1034, or error LNK1104, your developer command prompt is not set up correctly. For information on how to fix this issue, go back to the **Open a developer command prompt** section.
 >
 > If you get a different compiler or linker error or warning, review your source code to correct any errors, then save it and run the compiler again. For information about specific errors, use the search box at the top of this page to look for the error number.

7. To run your program, enter `hello` at the command prompt.

 The program displays this text and then exits:

 Output

   ```
   Hello, World! This is a native C program compiled on the command line.
   ```

 Congratulations, you've compiled and run a C program by using the command line.

Next steps

This "Hello, World" example is about as basic as a C program can get. Real world programs have header files and more source files, link in libraries, and do useful work.

You can use the steps in this walkthrough to build your own C code instead of typing the sample code shown. You can also build many C code sample programs that you find elsewhere. To compile a program that has more source code files, enter them all on the command line:

```
cl file1.c file2.c file3.c
```

The compiler outputs a program called *`file1.exe`*. To change the name to *`program1.exe`*, add an /out linker option:

```
cl file1.c file2.c file3.c /link /out:program1.exe
```

And to catch more programming mistakes automatically, we recommend you compile by using either the /W3 or /W4 warning level option:

```
cl /W4 file1.c file2.c file3.c /link /out:program1.exe
```

The compiler, cl.exe, has many more options you can apply to build, optimize, debug, and analyze your code. For a quick list, enter `cl /?` at the developer command prompt. You can also compile and link separately and apply linker options in more complex build scenarios. For more information on compiler and linker options and usage, see C/C++ Building Reference.

You can use NMAKE and makefiles, or MSBuild and project files to configure and build more complex projects on the command line. For more information on using these tools, see NMAKE Reference and MSBuild.

The C and C++ languages are similar, but not the same. The Microsoft C/C++ compiler (MSVC) uses a basic rule to determine which language to use when it compiles your code. By default, the MSVC compiler treats all files that end in *`.c`* as C source code, and all files that end in *`.cpp`* as C++ source code. To force the compiler to treat all files as C no matter the file name extension, use the /TC compiler option.

By default, MSVC is compatible with the ANSI C89 and ISO C99 standards, but not strictly conforming. In most cases, portable C code will compile and run as expected. The compiler provides optional support for the changes in ISO C11/C17. To compile with C11/C17 support, use the compiler flag **`/std:c11`** or **`/std:c17`**. C11/C17 support requires Windows SDK 10.0.20201.0 or later. Windows SDK 10.0.22000.0 or later is

recommended. You can download the latest SDK from the Windows SDK page. For more information, and instructions on how to install and use this SDK for C development, see Install C11 and C17 support in Visual Studio.

Certain library functions and POSIX function names are deprecated by MSVC. The functions are supported, but the preferred names have changed. For more information, see Security Features in the CRT and Compiler Warning (level 3) C4996.

See also

Walkthrough: Creating a Standard C++ Program (C++)
C Language Reference
Projects and build systems
Compatibility

Walkthrough: Compiling a C++/CX Program on the Command Line

Article • 03/01/2023

> **Note**
>
> For new UWP apps and components, we recommend that you use **C++/WinRT**, a standard C++17 language projection for Windows Runtime APIs. C++/WinRT is available in the Windows SDK from version 1803 (10.0.17134.0) onward. C++/WinRT is implemented entirely in header files, and is designed to provide you with first-class access to the modern Windows API.

The Microsoft C++ compiler (MSVC) supports C++ component extensions (C++/CX), which has additional types and operators to target the Windows Runtime programming model. You can use C++/CX to build apps for Universal Windows Platform (UWP), and Windows desktop. For more information, see A Tour of C++/CX and Component Extensions for Runtime Platforms.

In this walkthrough, you use a text editor to create a basic C++/CX program, and then compile it on the command line. (You can use your own C++/CX program instead of typing the one that's shown, or you can use a C++/CX code sample from another help article. This technique is useful for building and testing small modules that have no UI elements.)

> **Note**
>
> You can also use the Visual Studio IDE to compile C++/CX programs. Because the IDE includes design, debugging, emulation, and deployment support that isn't available on the command line, we recommend that you use the IDE to build Universal Windows Platform (UWP) apps. For more information, see **Create a UWP app in C++**.

Prerequisites

You understand the fundamentals of the C++ language.

Compiling a C++/CX Program

To enable compilation for C++/CX, you must use the /ZW compiler option. The MSVC compiler generates an .exe file that targets the Windows Runtime, and links to the required libraries.

To compile a C++/CX application on the command line

1. Open a **Developer Command Prompt** window. For specific instructions, see To open a developer command prompt window.

 Administrator credentials may be required to successfully compile the code, depending on the computer's operating system and configuration. To run the command prompt window as an administrator, right-click to open the shortcut menu for the command prompt and then choose **More** > **Run as administrator**.

2. Change the current working directory in the command prompt window to a directory you can write to, such as your Documents directory.

3. At the command prompt, enter **notepad basiccx.cpp**.

 Choose **Yes** when you're prompted to create a file.

4. In Notepad, enter these lines:

 C++

   ```
   using namespace Platform;

   int main(Platform::Array<Platform::String^>^ args)
   {
       Platform::Details::Console::WriteLine("This is a C++/CX program.");
   }
   ```

5. On the menu bar, choose **File** > **Save**.

 You've created a C++ source file that uses the Windows Runtime Platform namespace namespace.

6. At the command prompt, enter `cl /EHsc /ZW basiccx.cpp /link /SUBSYSTEM:CONSOLE`. The `cl.exe` compiler compiles the source code into an `.obj` file, and then runs the linker to generate an executable program named basiccx.exe. The /EHsc compiler option specifies the C++ exception-handling model, and the /link flag specifies a console application.

7. To run the basiccx.exe program, at the command prompt, enter **basiccx**.

The program displays this text and exits:

Output

```
This is a C++/CX program.
```

See also

Projects and build systems
MSVC Compiler Options

Walkthrough: Compiling a C++/CLI Program on the Command Line

Article • 02/24/2023

You can create Visual C++ programs that target the Common Language Runtime (CLR) and use the .NET Framework, and build them on the command line. Visual C++ supports the C++/CLI programming language, which has additional types and operators to target the .NET programming model. For general information about the C++/CLI language, see .NET Programming with C++/CLI (Visual C++).

In this walkthrough, you use a text editor to create a basic C++/CLI program, and then compile it on the command line. (You can use your own C++/CLI program instead of typing the one that's shown, or you can use a C++/CLI code sample from another help article. This technique is useful for building and testing small modules that have no UI elements.)

Prerequisites

You understand the fundamentals of the C++ language.

Compiling a C++/CLI Program

The following steps show how to compile a C++/CLI console application that uses .NET Framework classes.

To enable compilation for C++/CLI, you must use the /clr compiler option. The MSVC compiler generates an .exe file that contains MSIL code—or mixed MSIL and native code—and links to the required .NET Framework libraries.

To compile a C++/CLI application on the command line

1. Open a **Developer Command Prompt** window. For specific instructions, see To open a developer command prompt window.

 Administrator credentials may be required to successfully compile the code, depending on the computer's operating system and configuration. To run the command prompt window as an administrator, right-click to open the shortcut menu for the command prompt and then choose **More** > **Run as administrator**.

2. Change the current working directory in the command prompt window to a directory you can write to, such as your Documents directory.

3. At the command prompt, enter `notepad basicclr.cpp`.

 Choose **Yes** when you're prompted to create a file.

4. In Notepad, enter these lines:

 C++

   ```
   int main()
   {
       System::Console::WriteLine("This is a C++/CLI program.");
   }
   ```

5. On the menu bar, choose **File** > **Save**.

 You've created a Visual C++ source file that uses a .NET Framework class (Console) in the System namespace.

6. At the command prompt, enter `cl /clr basicclr.cpp`. The cl.exe compiler compiles the source code into an .obj file that contains MSIL, and then runs the linker to generate an executable program named basicclr.exe.

7. To run the basicclr.exe program, at the command prompt, enter `basicclr`.

 The program displays this text and exits:

 Output

   ```
   This is a C++/CLI program.
   ```

See also

C++ Language Reference
Projects and build systems
MSVC Compiler Options